AFRL-HE-WP-TR-2005-0006

United States Air Force
Research Laboratory

A High-Level Symbolic Representation for Intelligent Agents across Multiple Architectures

Jacob Crossman
Robert Wray
Paul Nielsen
Randolph M. Jones
Al Wallace

Soar Technology, Inc.
3600 Green Court, Suite 600
Ann Arbor, MI 48105

Christian Lebiere

Micro Analysis & Design
4949 Pearl East Circle, Suite 300
Boulder, CO 80301

July 2004

Final Report for the Period April 2003 to July 2004

Human Effectiveness Directorate
Warfighter Interface Division
Cognitive Systems Branch
2698 G Street
Wright-Patterson AFB OH 45433-7604

NOTICES

When US Government drawings, specifications or other data are used for any purpose other than a definitely related Government procurement operation, the Government thereby incurs no responsibility nor any obligation whatsoever, and the fact that the Government may have formulated, furnished, or in any way supplied the said drawings, specifications or other data, is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Please do not request copies of this report from the Air Force Research Laboratory. Additional copies may be purchased from:

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161

Federal Government agencies and their contractors registered with the Defense Technical Information Center should direct requests for copies of this report to:

Defense Technical Information Center
8725 John J. Kingman Road, Suite 0944
Ft. Belvoir, VA 22060-6218

TECHNICAL REVIEW AND APPROVAL

AFRL-HE-WP-TR-2005-0006

This report has been reviewed by the Office of Public Affairs (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

FOR THE COMMANDER

//SIGNED//

MARIS M. VIKMANIS
Chief, Warfighter Interface Division
Air Force Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
July 2004	Final	April 2003 - July 2004

4. TITLE AND SUBTITLE
A High-Level Symbolic Representation for Intelligent Agents across Multiple Architectures

5a. CONTRACT NUMBER
F33615-03-C-6343

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
63832D

6. AUTHOR(S)
Jacob Crossman, Robert Wray, Paul Nielsen, Randolph M. Jones, Al Wallace, Christian Lebiere

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
0476DM03

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Soar Technology, Inc.
3600 Green Court
Suite 600
Ann Arbor, MI 48105

Micro Analysis & Design
4949 Pearl East Circle
Suite 300
Boulder, CO 80301

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Human Effectiveness Directorate
Warfighter Interface Division
Cognitive Systems Branch
Wright-Patterson AFB OH 45433-7604

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-HE-WP-TR-2005-0006

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the High-Level Symbolic Representation (HLSR) project for the U.S. Air Force PRDA 03-01-HE: Human Performance in Modeling and Simulation, Technical Area 2: Opposing Force Behaviors. This report summarizes the work done on Defense Modeling Simulation contract F33615-03-C-6343 to develop a high level symbolic representation (HLSR) for behavior modeling. This effort seeks to increase development efficiency and reuse in behavior modeling. The report describes the development of a high level language that abstracts the details of individual intelligent system architectures (ISA), allowing developers to focus their effort on tasks directly related to producing intelligent behavior. This language is designed to be complied into executable representations on multiple ISAs. This report targets two ISAs, Soar and ACT-R. These ISAs have a proven tract record of generating capable behavior models in many domains. There were three primary goals. First, the desire to construct a specification for HLSR sufficient for its use in behavior modeling and for compiler design and implementation. Second, the desire to define mappings and probable transformation processes from HLSR to Soar and ACT-R. Third, the desire to prove the feasibility of this approach by demonstrating how HLSR solved real problems faced by knowledge engineers and how an HLSR model, which solved these problems, could be compiled to an ISA.

15. SUBJECT TERMS Intelligent System Architecture (ISA), Modeling and Simulation, Opposing Force Behaviors, Environmental Representation, Human Performance Modeling, Model Validation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Lt. R. Benjamin Hartlage
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFED	SAR	184	19b. TELEPHONE NUMBER *(include area code)* (937)255-9662

THIS PAGE LEFT INTENTIONALLY BLANK

Table of Contents

Acknowledgements

This report describes the High-Level Symbolic Representation (HLSR) project for the U.S. Air Force PRDA 03-01-HE: Human Performance in Modeling and Simulation, Technical Area 2: Opposing Force Behaviors.

We would like to thank our program and technical sponsors, Dr. Sheila Banks (AFRL/HECS), Dr. Mike Young (AFRL/HECS), Mr. Jim Anthony (DMSO), and Ms. Helen Redwine-Smith (AFRL/HEF) for their interest in and support of this project. We would also like to thank the government project managers, Lt. Randy Allen (AFRL/HECS) and Lt. Ben Hartlage (AFRL/HECS) for monitoring and administering this project. With the support provided by program and technical sponsors, we were able to develop a functional specification and prototype compiler design for HLSR.

We also thank our teammate and subcontractor, Dr. Christian Lebiere for his technical support of this project. Dr. Lebiere provided outstanding support to the project and valuable insight into the ACT-R cognitive model.

Finally, we would like to thank the entire Soar Technology team for the tremendous and successful effort they applied to this program. This program would never have achieved its goals without the considerable efforts of Mr. Jacob Crossman, Dr. Robert Wray, and Dr. Randolph Jones, as well as Mr. David Ray and Mr. Patrick Kenney who helped implement elements of the HLSR2Soar compiler. We also would like to thank Dr. Paul Nielsen, Mr. Jon Beard, Mr. Sean Lisse, and Mr. Glenn Taylor who helped form the basis of the concept we ultimately implemented during the execution of this program.

Executive Summary

This report summarizes the work done by Soar Technology, Inc. on Defense Modeling and Simulation contract F33615-03-C-6343 to develop a high level symbolic representation (HLSR) for behavior modeling. Large-scale, capable behavior models are very difficult to build. They require significant effort from highly trained experts in behavior modeling and intelligent system architectures (ISA). ISAs are the preferred systems on which to build behavior models because they define unique approaches to computation, enabling intelligent, flexible, autonomous software that can approximate the richness and sophistication of human behavior. However, programming within these architectures is tedious and costly. Further, direct reuse of program elements is rare. Programmers must manage the internal processes of the architectures at a fine-grained level. Because the programs encode details of architecture, transfer of domain knowledge across behavior models is presently impractical. These factors all increase cost.

This effort seeks to increase development efficiency and reuse in behavior modeling. We describe the development of a high level language that abstracts the details of individual ISAs, allowing developers to focus their effort on tasks directly related to producing intelligent behavior. This language is designed to be compiled into executable representations on multiple ISAs. We are targeting two ISAs, Soar and ACT-R. These ISAs have a proven track record of generating capable behavior models in many domains.

We had three primary goals. First, we desired to construct a specification for HLSR sufficient for its use in behavior modeling and for compiler design and implementation. Second, we desired to define mappings and probable transformation processes from HLSR to Soar and ACT-R. Third, we desired to prove the feasibility of this approach by demonstrating how HLSR

solved real problems faced by knowledge engineers and how an HLSR model, which solved these problems, could be compiled to an ISA.

To achieve these goals we researched and documented commonalities and differences between ISAs, focusing on Soar and ACT-R. We also documented common problems and solutions faced by knowledge engineers as they build and maintain behavior models. Based on these results, we designed the HLSR language, and developed a formal specification of the language. We defined mappings from HLSR to ACT-R and Soar that enable the compilation of HLSR models to native representations. We developed micro-theories for the execution of HLSR constructs in Soar, and in order to facilitate effective compilation, an ontological model of Soar that describes the primitives and constraints of the Soar architecture. Finally, we conducted a feasibility test by designing and implementing a simple behavior model in HLSR. We then hand compiled this example using the HLSR mappings to Soar and ACT-R.

This effort provides important advances. First, the specification for HLSR is defined in sufficient detail to enable the development of behavior models in this language. HLSR is independent of any ISA, and is thus unique among behavior representations. Second, we have shown both at the conceptual level and at the implementation level that HLSR models can be compiled to both Soar and ACT-R. Our analysis suggests HLSR addresses many of the issues that make behavior development inefficient and difficult. Critically, ISA-independence enables the abstraction of ISA details that distract from the modeling process and require specialized ISA training. Furthermore, the HLSR specification (and the supporting theoretical results) provide a mature foundation for full scale compiler development.

THIS PAGE LEFT INTENTIONALLY BLANK

1 Introduction

1.1 The Problem and the HLSR Solution

Currently, intelligent systems are encoded in knowledge representations specific to particular intelligent system architecture (ISA). Examples of such architectures include Soar [30, 38, 48] and ACT-R [2, 3], both of which have been used to model complex human decision making and behavior tasks. For example, ACT-R and Soar models have been developed to model performance and learning within a simulated air-traffic control task [10, 29] and both architectures have been used to implement embodied behavior representations for urban combat training [6, 50]. This process of encoding knowledge for execution on an ISA is typically carried out by highly trained knowledge engineers and is tedious, time consuming, and error prone. Thus model development is a costly process.

There are several specific problems with the traditional approach. First, a knowledge engineer must be trained in the many subtleties and implementation details of the specific architecture for which they are building the model. Such expertise is difficult (and thus expensive) to obtain. Furthermore, since most knowledge engineers specialize in a single ISAs, they cannot take advantage of the capabilities other ISAs. Second, significant effort in building a behavior model is devoted to managing details of the implementation, rather than adding capabilities to the model. This effort describes some of these problems in section 2.2.1 and Appendix A, which we have discovered are largely shared by both Soar and ACT-R. Third, partly as a result of the first and second problems, behavior models are very difficult to reuse. Reuse of domain specifications across architectures is non-existent, and, surprisingly, reuse is rare even within an architecture. Reuse between architectures is hindered both by the

architectural details encoded within the model and by the differing design approaches encouraged by the architectures. Reuse within an architecture is hindered by the following:

1. Different design approaches to managing architecture details, referred to "idioms" [28]
2. A lack of architectural support for engineering tasks such as knowledge encapsulation
3. Varying levels of expertise between knowledge engineers

Together, these problems make architecture-based models expensive to build and maintain, difficult to reuse, and limited in capability and robustness.

Our long-term objective, of which the HLSR is a critical foundational component, is to create ISA-independent higher level behavior modeling languages and tools that hide architectural details and provide higher-level abstractions. We believe that these languages and tools will enable the following:

1. Behavior developers to spend more time encoding task directed knowledge
2. Tool developers to build better end-user tools for viewing and modifying behaviors

The convergent computational properties of ISAs facilitate this goal. Despite starting from different intellectual traditions (e.g. artificial intelligence, cognitive psychology, and even philosophy), knowledge-intensive, embedded ISAs increasingly demonstrate quite similar computational approaches to intelligent behavior [23]. We describe these commonalities in sections 2.3.

This convergence in computational approach provides the opportunity to define high-level languages that can describe behaviors on multiple ISAs. In this effort, we defined one such language that we refer to as the high level symbolic representation (HLSR). It is high-level in that its constructs are ISA independent and more abstract than the ISA equivalent structures. It is symbolic in that its constructs and constraints depend on knowledge being represented

symbolically. This HLSR will allow knowledge (agent programs) specified at the HLSR level to be executed on a variety of target ISAs.

The situation is analogous to the emergence of the high level languages such as FORTRAN, C++, and Java in computer science. These languages enabled the same development improvements that we seek with HLSR. Namely, they provided architecture independence; programs no longer needed to be rewritten for each new architecture. Furthermore, they abstracted the software developer from the details of architecture implementation and provided them with constructs that mapped well to programming tasks and computer science theory. As these languages have developed, they have abstracted more of the tedious details of programming and enabled larger and more complex systems to be developed.

We see HLSR as an early step in a similar process for ISAs. With HLSR we seek to put a first layer of abstraction on top of ISAs and prove the feasibility of compilation to multiple ISAs. We intend that this effort will lead to further efforts to improve HLSR and to build related development tools and domain-specific representations that leverage HLSR.

1.2 Related Work

As discussed above, the idea of high-level languages is not new, and it is not particularly novel for intelligent system architectures either. This work shares many similarities with attempts to build higher level abstractions within the specific architectural paradigms of Soar and ACT-R. For example, the Task Acquisition Language (TAQL) [51] was developed to allow Soar programmers to specify knowledge at the level of Soar's basic computational theory (the Problem Space Computational Model), abstracting some of the theoretical implementation details necessary for model implementation. More recently, ACT-Simple [39] has been

developed to convert models specified at the high level of GOMS[1] to ACT-R, resulting in models that better match human error and that can be developed much more quickly than comparable ACT-R models [21]. Another GOMS-to-ACT-R compiler, G2A, has also been developed [40].

While both TAQL and ACT-Simple enable rapid specification of simple models, both approaches make assumptions that limit their utility for their *general* application to intelligent systems. TAQL focused on operator representation in Soar, which is the most strongly principled Soar representation. However, TAQL did not provide much guidance in the elaboration and maintenance of beliefs, which has proven extremely important in the development of Soar applications [22]. This incompleteness meant developers still often needed to work at the Soar level, defeating many of the advantages of a high level language. ACT-Simple is tied to the assumptions of GOMS. GOMS covers only a narrow range of expert human tasks, such as pressing keys and moving a mouse and does not make a commitment to many elements needed for knowledge-intensive intelligent systems [23]. While ACT-Simple uses many of ACT-R's mechanisms to cover those missing in GOMS, it does so by making assumptions about domains and tasks within those domains, limiting its applicability to the problem of the representation of knowledge for intelligent systems that span many different application areas. Finally, both of these solutions are specific to a particular architecture. A crucial innovation of HLSR is that it spans architectures, which will not only lead to reuse of model components across architectures but also lower barriers to entry for the development of intelligent systems.

---

[1] Goals, Operators, Methods, and Selection Rules. GOMS is a modeling technique developed and used primarily for modeling human computer interactions.

Attempts to solve the behavior development problem described above need not include high level languages. Many approaches attempt to speed behavior development via better development tools. The tools can be divided into two broad categories: integrated development environments and training behavior systems by observation.

Development environments are a necessary component for efficiently programming any system. Both ACT-R and Soar have such environments, and work continues to improve them. However, while development environments make engineers more efficient, they do not change the level at which engineers do their work. Therefore, while some efficiency improvements can be achieved with integrated development environment (IDE) improvements, significant changes in development speed and cost are not likely to be enabled by these tools. In traditional software engineering, the biggest gains in development speed have been achieved through improved methodologies and abstractions (e.g. object-oriented programming, component methodologies, and languages). While tools can support the engineer in using these methodologies and abstractions, tools are only as good as the representations and abstractions they present to the user.

Training by example and visual programming systems attempt to reduce behavior modeling to series of examples that show an agent how to react in various situations [4, 12, 20, 34, 36, 44]. This approach offers the potential to allow end users to "train" intelligent systems. The cycle of knowledge transfer from subject matter expert (SME) to knowledge engineer (KE) to end user is broken. Development costs are cheaper, the end user is more likely to get what they want, and the behavior model can be continuously improved over time. However, many technical challenges must be resolved before this approach becomes feasible. Currently, such systems fall far short of the specified goal and operate very much like interactive development

environments. One important reason fine-grained iterative refinement is required is the very low level abstractions and structures on which the tool is built. That is, the intelligent system learns at the level of architecture primitives rather than higher level behavior. The user is then forced to interact directly with the terms and idiosyncrasies of the underlying architecture on which the behavior will ultimately execute. If this project is successful, HLSR may be a better language target for learning by observation systems than the low level languages of ISAs.

Tools such as development environments and learning by example systems are valuable, but both could benefit from the higher level, cross-platform abstraction that HLSR provides. HLSR would provide a common foundation for these tools, assuring that they had a common methodology and structure behind them, and allowing knowledge developed using these tools to be shared across different architectures.

1.3 Document Organization

This document is primarily organized around the technical objectives of this effort as given in the statement of work. These objectives include (slightly reworded for clarity):

1. Research and analyze HLSR development requirements
2. Research and analyze the requirements of cognitive architectures
3. Design specification language
4. Design low-level behavior primitives
5. Create reference model for target architecture
6. Design code generation capability for cognitive models
7. Demonstrate feasibility of the approach

Sections 2.2, 2.3, and 2.4 covers (1) and (2), section 2.5 covers (3) and (4), section 2.5 covers (5) and (6), and finally, section 3 covers (7). For each technical objective we describe our

process and the technical outcome. The technical outcome of some objectives is a separate report or document, which are referenced as appropriate. Final results and evaluation is left to section 4. Here the results are discussed along with the answers to the key research questions of the effort.

2 Research and Development

2.1 Research Questions and Methodology

This section introduces the research questions that motivated the work and describes the methodology used in the execution of the work.

2.1.1 Research Questions

While other higher level representations have been built (section 1.2), none to our knowledge has attempted to apply across multiple ISAs as HLSR does. Because this ambition is novel, fundamental questions about the feasibility and character of the HLSR solution abound. A significant portion of this effort focused on answering these questions, which can be categorized into four board categories:

1. To what extent can a single language be used across multiple ISAs? More specific research questions include: I
    - Is cross ISA compatibility technically possible (given current methodologies, tools, and knowledge of ISAs)?
    - If possible, how effectively will HLSR leverage the capabilities of each ISA
    - How similar are ISAs? What are ISA commonalities and differences?
    - Can abstractions be designed that take advantage of similarities?
    - Can the differences be hidden from the developer and managed exclusively by the compiler?
2. What high level constructs are needed to support programmability and ease-of-use?

3. Are behavior systems compatible with traditional methods of encapsulation, and is encapsulation the best method to improve the reuse of knowledge representations?
    - What aspects of behavior can be encapsulated and reused?
    - What constructs and processes are necessary for encapsulation and reuse of behavior?
    - What aspects of behavior cannot be encapsulated effectively because of ISA constraints and the processes involved with intelligence?
4. Can an HLSR representation be compiled to an ISA representation and to what extent is performance efficiency compromised by compilation?

High level constructs and programmability are well understood in computer science and has been effectively implemented in SE languages and tools, but have not been successfully implemented in behavior modeling. However, for the sake of ease-of-use, high-level behavior representations have tended to abstract away some of the details necessary to construct a behavior model; that is, they have not been *complete*. Further, ISAs tend to support runtime adaptation and learning, and changes to knowledge at runtime have not been easy to map back into the higher level abstraction. For HLSR, these issues lead to two fundamental questions: "Can HLSR be complete and still high-level?" and "Can learned knowledge be merged with HLSR knowledge without requiring the engineer or end user to understand the details of the ISA executed the behavior?"

Encapsulation and reuse are also common elements of traditional SE. In fact, traditional SE has made significant, though incomplete, strides in this direction for the past few decades. However, behavior modeling is resistant to encapsulation. Is it because it is fundamentally impossible to decompose and encapsulate behaviors into reusable modules? Or is it because knowledge engineers tend to focus their engineering efforts on different aspects of the problem

other than encapsulation and reuse? We believe that it is fundamentally *difficult* to decompose behaviors due in part to the complex dependencies they require. For example, determining and understanding the actual modularity in human behavior is a long-standing research issue and source of significant contention in cognitive psychology [14].

Compilation process and efficiency is the most practical of the issues addressed in this effort. These issues are primarily engineering issues, and as such are not a primary focus of this effort. However, we are interested in two fundamental question regarding compilation and efficiency that directly effect the feasibility of our approach. First, we would like to answer the question "Can a compilation process be defined that correctly transforms HLSR constructs to ISA constructs?" Second, we would like to answer the question "Are there any fundamentally intractable problems in compilation or execution of compiled knowledge?" If the answers to these questions are "yes" and "no" respectively, we are confident that incremental engineering advancements can improve efficiency over time.

All of the questions have been addressed within this project effort; however, some require continued efforts such as full-scale compiler implementation and case studies to answer effectively. Section 4.1.3 summarizes our answers to these questions given the research and development in this effort.

2.1.2 Research Methodology

Our methodology centers on the computer science concept of "proof by construction." We proposed to build an HLSR and show how it can compile to two ISAs – Soar and ACT-R.

To answer the questions related to questions about cross ISA compatibility, we researched and documented the similarities and differences between ISAs, in particular between Soar and ACT-R. The outcome of this process is summarized in section 2.3. To answer

questions related to high-level constructs and programmability, we constructed a catalog of common patterns and solutions in behavior development discussed in section 2.2.1. We then attempted to develop solutions in HLSR that reduce or eliminate the knowledge engineering effort necessary to manage these patterns. To answer questions related to encapsulation and reuse, we analyzed traditional SE approaches to encapsulation and reuse, and selected those that best integrated with ISAs for inclusion in the HLSR specification. The results of this effort are principles of behavior model development (section 2.2.2) and core HLSR primitive constructs for encapsulation (see 2.5.1).

To answer questions related to compilation process and efficiency, as well as the overarching question of HLSR feasibility, we conducted a feasibility demonstration. The details of this demonstration are given in section 3. The feasibility test included selection of a small problem then walking through the design, encoding, compilation, and (in the case of Soar) execution phases using HLSR. The compilation process is still in its primitive stages and for our feasibility demonstration was done by hand. However, a limited functionality prototype compiler has been implemented for Soar (described in 3.2.6).

2.2 Research and Analyze HLSR Development Requirements

An initial task of this project was to understand the issues related to the intelligent system design and development process in sufficient detail to design solutions. To accomplish this goal we analyzed intelligent system programming tasks for patterns of problems and solutions. This catalog informed the HLSR requirements discussed in section 2.4 and led to principles of development that constrain the HLSR specification.

2.2.1 Catalog of Common Problems and Solutions

The catalog of common problems and solutions summarizes the common patterns of problems and solutions a Soar or ACT-R developer typically faces and manages when developing a behavior model. These patterns describe in detail the difficulties of building and maintaining behavior models. Furthermore, the HLSR specification defines specific constructs, constraints, and processes to help make these patterns easier or unnecessary to implement.

We divided the catalog into three sections as follows:

- Catalog of Low-level Details HLSR Should Abstract
- Catalog of High-level Tasks HLSR Should Make Easier
- Catalog of Micro-Patterns That Developers Typically Use

Developers from the HLSR development team each contributed a description of two or more common patterns from their own experience building and maintaining behavior models for Soar and ACT-R. The complete catalog is provided as Appendix A. Here we summarize the catalog in development. Table 1 below.

Problem/Pattern	Description	Impact on Development
Catalog of Low-level Details HLSR Should Abstract		
Process tagging	"Process done" and "process status" tags to track the state of processes that manipulate knowledge structures	Process specific structure gets "tangled" with structure that is intrinsic to objects, leading to maintenance problems.
Logic Tricks	Using logic axioms to simulate missing logical primitives (e.g. "or")	Developers use obscure logic tricks to implement the missing logic operators. Difficult to understand, debug, and maintain.
Copying and Memory Manipulations	Creating copies of an object as well as creating special case knowledge for managing different forms of copying (e.g. replacement v. creating from scratch).	Developers spend time writing structure specific copy code, many times repeatedly for different contexts. This wastes development time and maintenance costs increases because each point of copy has to be maintained.

Detailed Structure Specification	Managing the details of knowledge structures such as layout, decomposition of larger structures to smaller structures, and linking different structures together.	Developers spend a lot of time twiddling with the details of knowledge structure, rather than solving the task of interest.
Catalog of High-level Tasks HLSR Should Make Easier		
Planning	Managing the details involved with planning such as creating and maintaining imaginary states, executing projections, tracking and modifying plans, etc.	Developers either don't use planning processes, or create their own, usually limited process from scratch for each application.
Writing production code procedurally	Specifying the processes that are being carried out as a series of procedural constraints, leaving conversion to productions to the compiler	Developers encode constraints and dependencies in a set of productions. These dependencies are implicit and hard to understand and track leading to difficulty in debugging and maintenance.
Knowledge Integration	Integrating and reusing knowledge from different models, representations, and ontologies	Developers rarely reuse knowledge within an architecture and never reuse knowledge directly between architectures.
Implicit Semantics	Knowledge depends on semantics that are implicit in the representation. These semantics should be made explicit wherever possible.	Developers consistently misunderstand the structure and intent of knowledge during development and maintenance leading to errors and time consuming analysis of knowledge.
Goal Manipulation	Managing goals and relationships between goals in a way that abstracts the details of how the goals and their relationships are stored and how goals are retrieved.	Developers write code that manages the details of structuring and retrieving goals in a goal hierarchy. Most of this code is tedious and orthogonal to the behavior of interest.
Perceptual Motor Interaction	Managing and manipulating sensory/motor processes in a simple way, abstracting how these processes are done within the architecture.	Each developer manages and manipulates sensory motor information in different ways making systems difficult to reuse and maintain.
Catalog of Micro-Patterns That Developers Typically Use		
Retrieve v. Compute	Dual logic for either execution of a process or retrieval of a previously computed answer gets repeated with process-based variations.	Each situation is handled independently. Depending on the process and developer experience, different retrieve/compute strategies may be implemented within the same behavior model.

Output command structures & Proprioception	The pattern of creating output commands of one of a few types and managing the feedback from the motor system as to its progress. This is related to "Perceptual Motor Interaction" above, but speaks more of a specific set of processes to abstract.	Sophisticated behavior code must be developed to manage the execution of motor commands. Many bugs are introduced because of subtleties in motor system interaction that are not apparent to developers who have not worked with a particular type of environment. End result is often a simplistic motor system that barely achieves functional requirements.
List Management	Managing collections of information in either sequential or non-sequential form.	Many different implementations of the same list management processes. List management data gets tangled with other knowledge structures as with "process tagging" above.
Iteration	Process a series of objects in some order while not losing reactivity	Many different implementations of iteration processes. Many "clever" implementations seek to optimize performance or the amount of code required, but obscures the purpose of the process.
Understanding when a process involving multiple objects is complete	The patterns of processes that are commonly used to determine when all of a collection of objects have been processes in some way.	Complex logic to simulate universal quantifiers is added or "aspect-like" (in the aspect oriented programming sense) processes are used to monitor progress and flag completion. In either case the code can be difficult to understand and maintain.

Table 1: Summary of catalog of common problems and solutions in behavior development.

We describe one pattern in detail here to illustrate our approach to using these patterns. Consider the "process tagging" pattern. The process tagging pattern is a direct result of the reactive nature of intelligent systems. In an intelligent system, at any given time the process that is currently executing may be interrupted by some other process. Furthermore, processes are often structured such that they apply reactively in a broad range of contexts rather than a few narrow contexts. Because of these reactivity constraints, behavior model processes must store knowledge about the current processing status in a location independent of the process, so it is properly retained even if interrupted or started from a different context. The usual solution is to

augment knowledge structures with additional information about the processes execution on the knowledge structure. Figure 1 shows the problems caused by process tagging, and how HLSR can help solve these problems.

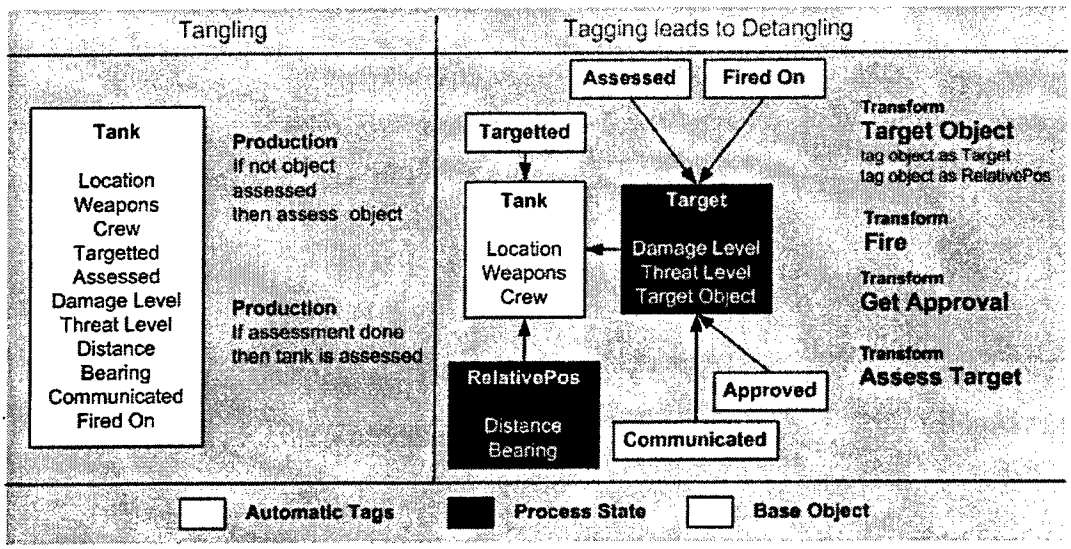


Figure 1: Process tagging and how HLSR can help abstract the details.

In Figure 1 above the Tank object has three intrinsic properties: location, weapons, and crew. However, when building processes that operate on the tank, the behavior developer begins to augment this structure with threat information, targeting information, damage levels, and relative location. Each of these pieces of information is important to the processes operating on the tank, but they have three detrimental effects:

- They clutter the tank definition with knowledge that is not intrinsic to the tank structure itself. We say that the process data is *tangled* with the intrinsic data. This makes understanding the tank object difficult for developers maintaining the system.
- The management of these tags is tedious and error prone. Processes often include common sense rules like "if not object assessed, then asses object." These types of

control logic are common across a large body of processes, and abstraction should be possible.

Over time, as more processes are added and removed from the system, some of the process tags are not removed, increasing clutter. It is often difficult to know exactly which tags are no longer used.

HLSR provides solutions for these problems that at minimum reduce the detail management and maintenance required to manage process tags. The approach is as follows:

- HLSR defines and refines the concept of tags. HLSR provides a memory manipulation primitive called "tag." A tag is an object that extends what is known about another object without breaking the structure intrinsic to the object being tagged. For example, in Figure 1 the Target object contains process information that is used by the Assess Target and Fire processes. Rather than augment the tank object with this process information, an explicit tag is created with the process relevant information and assigned to the tank using the special relationship "tagging." The process-specific information is no longer tangled with the tank's intrinsic properties.
- HLSR defines two built in relations called "tagging" and "tagged." These relationships are the inverse of each other. In Figure 1, the Target is information is said to be "tagging" the Tank, while the Tank is said to be "tagged" by the Target. HLSR provides built in keywords for testing for this relationship in rules and developer defined processes.
- HLSR automates some of the most tedious and generic tagging. In particular, HLSR provides constraints regarding process status. Because of these constraints, it is always possible to know whether a process has completed or not. HLSR defines a

special tagging process that automatically tags each goal with any transforms that have executed to achieve that goal. This alleviates the need to write the common sense rules described earlier – the compiler writes them for you.

This is just one example of a common pattern and solution that HLSR facilitates for a developer. Section 0 discusses how HLSR addresses the rest of the patterns described in the catalog.

2.2.2 Development Principles for Behavior Modeling

One result of our research on development processes and developer requirements was a set of development principles for intelligent systems. Each of these borrows to some extent from traditional SE methodologies and principles; however, they each have different applications and constraints specific to behavior modeling. These principles are important because they serve as the fundamental underpinnings for the HLSR design and specification. Many HLSR constructs and constraints are intended to specifically enforce or support one or more of these principles. Furthermore, these lead to a more methodological behavior development process, which is important for achieving more efficient behavior development across a wide range of developers and applications. This is similar to the way structured programming and object-oriented methodologies have improved traditional software engineering processes.

2.2.2.1 Least commitment to Dependencies

The principle of **least commitment to dependencies** states that process knowledge should depend on exactly the knowledge that it requires to execute the process – no more and no less. Though seemingly a simple and obvious principle, it is violated regularly in behavior

models. An example follows in Figure 2. Here the process "Plan Route" requires the amount of fuel available in order to execute successfully.

On the left we see *over commitment through excessive detail*, where the "Plan Route" process references each of the fuel tanks of the vehicle in order to calculate the total fuel available. This is over commitment because the planning process now depends implicitly on the number of fuel tanks in the vehicle. If an attempt is made to reuse this behavior in a vehicle with only one fuel tank, the process will fail, and the reason for the failure may be difficult to determine.

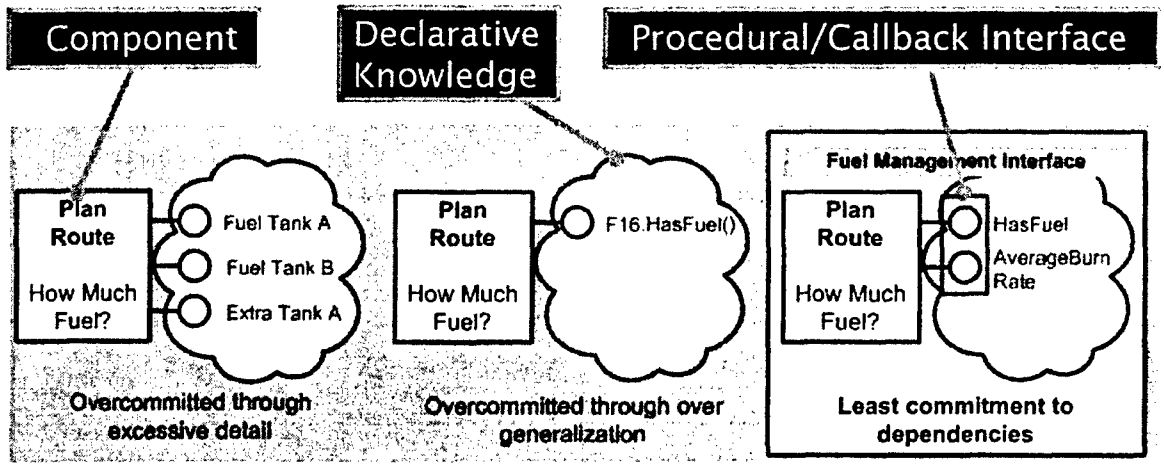


Figure 2: Diagram showing different levels of commitment to knowledge structure.

In the middle we see over commitment through over generalization, where the "Plan Route" process refers to the type of the vehicle (e.g. F16 aircraft) for which the fuel is being calculated. This is over commitment because the planning process depends on the existence of a whole class of information (that associated with F16) that is not relevant to the process itself. Again, if an attempt is made to reuse the process for another vehicle, the process will fail.

On the right, we see *least commitment to dependencies*, where the "Plan Route" process refers to exactly the information it requires for execution. It does this by referencing a

procedural or *callback* interface that defines exactly the processes that are required for execution. This process can be reused for other vehicle by simply instantiating the fuel management interface properly for the new vehicle. While the other two examples could be modified in a similar way, the modification for the least commitment example is well defined and explicitly called out. This idea is not entirely new, and is implemented at some level or another in traditional software systems for decades. However, current ISAs do not have mechanisms to support these types of abstractions.

HLSR provides constructs that directly support the principle of least commitment to dependency. These constructs include *explicit declarative structure* definitions which allow both the developer and compiler to know the scope of declarative structures such as vehicle descriptions, and *procedural interfaces* which provide direct support for the type of interface described in the example above.

2.2.2.2 Encapsulation

The principle of **encapsulation** states that problems are decomposed into knowledge units. These units hide the details of the solution, exposing well defined interfaces to other knowledge units. This principle is the foundation of object-oriented programming, but is difficult to translate to behavior models for the following reasons:

- ISAs encourage strong and often fluid interdependencies between knowledge to ensure that an appropriate decision is made for each context.
- Behavior model knowledge frequently changes structure. This is especially true for behavior models that learn.

These issues have led many behavior modelers to believe that encapsulation is undesirable, or is unattainable. However, if encapsulation is not used, behavior models quickly

reach a level of complexity that cannot be managed by developers, just as procedural software systems did before the advent of object oriented programming.

The desired result is to enable modular problem decomposition and detail hiding for the developer, while retaining the flexibility required for intelligent behavior. HLSR seeks to achieve this balance with the set of constructs defined below in Table 2.

Construct	Encapsulates	Exposes
Declarative Knowledge Encapsulation		
Declarative Object	Intrinsic relations (attributes)	Initializer, developer defined relations and interfaces
Goal	Structure, met logic	Met condition, developer defined relations and interfaces.
Process Encapsulation		
Manipulator	Transform Logic	Objects it manipulates (parameters)
Production	Reactive process initialization	Objects it manipulates (parameters)
Query	Condition logic	Objects it manipulates (parameters)
Mixed Process and Declarative Knowledge Encapsulation		
Production Set	Productions and related context	Developer defined relations and interfaces. Productions can react to global memory changes.
Transform	Local memory, transform logic	Associated goal, developer defined relations and interfaces.

Table 2: List of encapsulating constructs in HLSR.

HLSR's high-level constructs serve as the core encapsulation mechanisms in HLSR. It provides both process and declarative knowledge encapsulation. These are related to object and method encapsulation in object oriented systems, but differ as follows:

1. Transform constructs have declarative representations. This means that some of the process specific state can be carried along with the process, and that processes can be inspected for status by other processes – both important in intelligent systems.

2. Declarative knowledge structure is allowed to vary during execution. In traditional object-oriented systems this would be equivalent to class modification during execution. Furthermore the structure supplied by HLSR is treated as an execution constraint, not a hard requirement. Knowledge that violates encapsulation is allowed, but results in error conditions. For example, a behavior model can record that a tank has legs, but this will result in the behavior model becoming aware of an inconsistency between its standard model of a tank (with tracks) and its current instance (with legs).

These two key differences with traditional object-oriented encapsulation help to alleviate the first of the key problems with encapsulation in behavior models. That is, the fluidity of structure and relationships. The second problem is partially alleviated by the fact that HLSR does not require a specific structural representation of knowledge at the ISA level. That is, the ISA can define any low-level structure appropriate for representing a higher level HLSR construct. Because of this, the behavior model can learn new low level structures without being required to map them back into HLSR. The remaining issue is to understand how to map back learned knowledge into HLSR high-level structures. We have not yet resolved this issue as discussed later in section 4.4.

2.2.2.3 Explicit Declaration of Intention

The principle of **explicit declaration of intention** states that knowledge semantics designed by the developer should be explicitly encoded. That is, if a developer makes an important design decision, that decision should be obvious and reflected in the encoded knowledge.

In systems today, many relationships between processes and knowledge structures are left implicit. This problem is captured in Appendix A under "Implicit Semantics." Figure 3 below shows some of the advantages obtained by explicitly declaring intention.

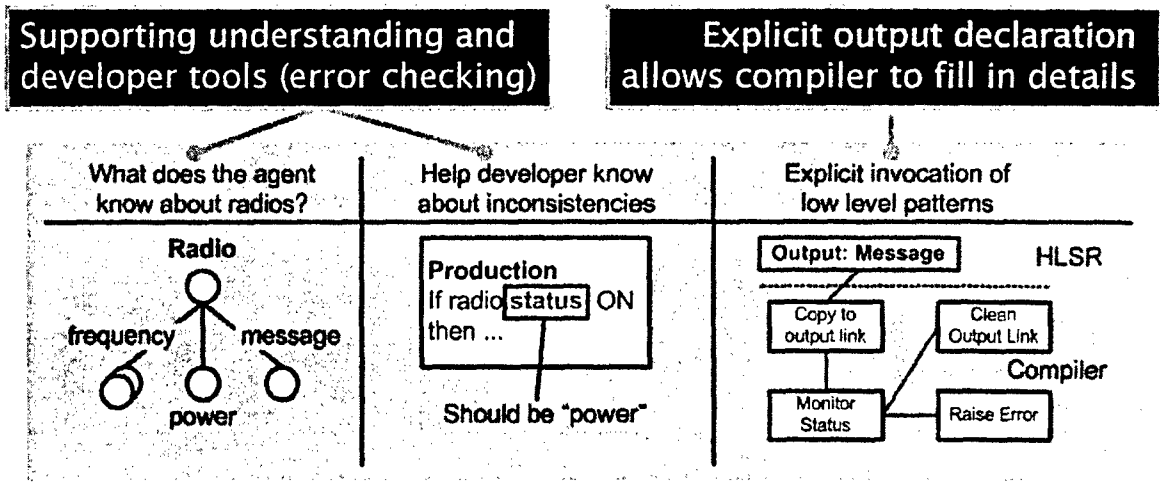


Figure 3: Ways in which intention can be made explicit.

HLSR supports explicit declaration of intention in several ways. First, HLSR is a typed language; that is, all declarative knowledge must have explicitly declared structure. While this structure is modifiable at runtime, any modifications must also update the type information about the structure being modified. Because of this both developers and executing behavior models can quickly understand what types of things a behavior model knows about. Furthermore, the compiler can help the developer by detecting errors in knowledge structure, which traditional software engineers have been taking advantage of for many years. Second, HLSR provides keywords to encode important and common relationships between procedures. Such relationships include sequential dependencies, and iteration patterns. Third, HLSR provides explicit, structured models for sensory motor interaction. By making these explicit and constraining them, the compiler is able to encode many of the details that developers previously encoded themselves.

To summarize, explicit declaration of intent improves developer understanding of the code, thus reducing maintenance costs; improves error checking, thus reducing bugs; and, enables the compiler to manage details by taking advantage of known semantics to fill in the low-level elements of a explicitly declared process.

2.2.2.4 *Abstract Low Level Details*

The principle of the **abstraction of low level details** states that the amount of knowledge that is not directly related solving the agent's domain tasks should be minimized. This principle indicates that low level programming details like those captured in the *Catalog of Low-level Details HLSR Should Abstract*, should be abstracted by a higher level representation. By *abstracting* we mean that the higher level representation should not require those details to be managed. Instead those details are managed by the architecture or the compiler.

Part of the solution for abstracting these details lies in our concept of "micro-theories for compilation" as described in Section 3.2.5. The concept of these micro theories is to capture the invariant aspects of these low level details and provide templates in the compiler for encoding them. The variant aspects of these details (e.g. the parameters) are then exposed for developer manipulation through HLSR or through compiler options. An example follows in Figure 4, which shows how two low level details – tagging and complex logic – are abstracted in HLSR.

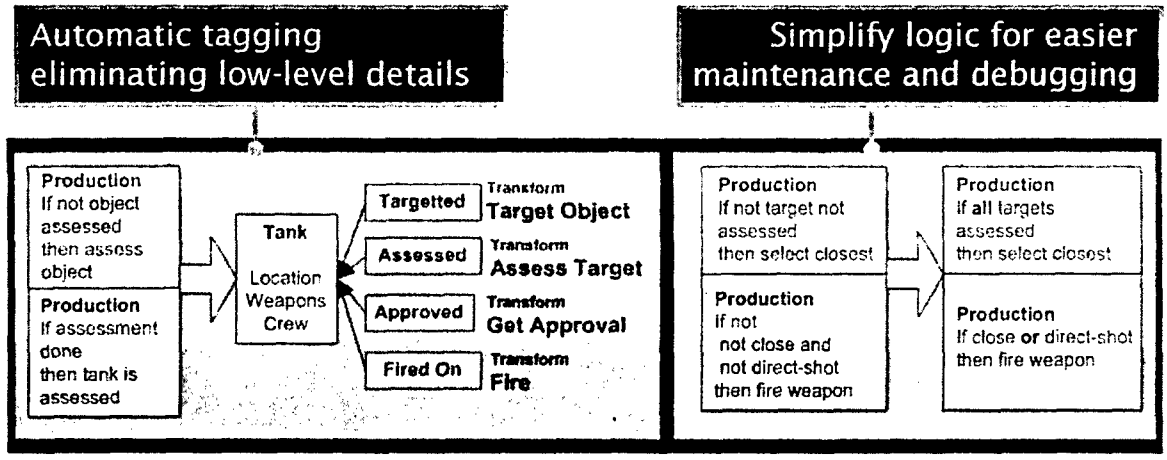


Figure 4: Abstracting tagging and complex logic.

Tagging is discussed in section 2.2.1 and in the *Catalog of Low-level Details HLSR Should Abstract*. Invariants of the tagging process include the structure of tags and their relationship to the object being tagged, and common status tags used to track goal-driven behavior (see 2.3.3.3). In Figure 4 we show that HLSR can standardize the structure and relationships of tags (Targeted, Assessed, etc) with the tagged object (Tank). The HLSR compiler can also generate code templates to perform common processes such as insuring that the same process is not re-executed on an object. In general, HLSR defines a process for automatically tagging goals with the transforms that attempt to achieve the goals, and it requires goals and transforms to be tagged with status indicators such as *executing*, *met*, *completed*, and *suspended*. Both of these tasks are managed by developers in Soar and ACT-R.

Complex logic is discussed in the *Catalog of Low-level Details HLSR Should Abstract*, as well. Often ISAs implement the few logical operators required to achieve functional completeness, with perhaps, one or two extras. The principles of logic define conversions and mapping that can be used to generate the rest. However, conversion logic is much harder for developers to interpret, and is fairly simple to generate with a compiler. Figure 4 shows the example of universal quantifiers, which map to the logical statement "is S true for all X," and the

logical OR operators. The current HLSR specification does not support universal quantifiers directly because of the difficulty of implementation in ACT-R (see *Understanding when a process involving multiple objects is complete* in the *Catalog of Micro-Patterns That Developers Typically Use*). However, HLSR does support iterative loops to achieve the same result as universal quantifiers with a more direct mapping to ACT-R. HLSR also supports more logical operators, such as OR and exclusive-or (XOR), than either Soar or ACT-R does natively, which improves code readability.

2.3 Research and Analyze Cognitive Architectures

The requirement to compile HLSR to both Soar and ACT-R representations, as well as potentially other ISAs in the future, led us to do a comparative analysis of ISAs and how they represent and manipulate knowledge. The result was a series of reports and a summary analysis of the commonalities and differences among architectures[2]. Of most interest to us were the commonalities, as they form the foundations for HLSR primitives. Analysis of architecture differences resulted in the principle of architecture discretion, discussed in 2.3.4.

ISAs share many similar structural components, and also process them similarly. We have noted [23] that each structure in an ISA resides in one of three general states, which we refer to as the *CCRU states* (see 2.3.3.2). The possible states are *latent* (inactive and possibly not yet existing), *considered* (decision required for activation), and *activated* (useful in current context). This is the result of the context driven processes necessary for least commitment to an execution path (see 2.3.3.1) and reactivity (see 2.3.3.4). In our summaries below we discuss how these

[2] All reports will be included (as separate files) in the electronic submission of this document.

states map to each structure in ACT-R and Soar. Section 2.3.3.2 discusses these states from a more abstract perspective, including the processes that change states.

2.3.1 Leveraging Cognitive Architectures

ISAs, and in particular cognitive architectures such as Soar and ACT-R, have important computational differences when compared to other computational systems such as the Von Neumann [2, 11, 30, 45]. These differences allow ISAs to execute behavior that is more complex, more adaptive, and more appropriate for a rich set of contexts than the behaviors produced by other computational systems. One goal of the HLSR design is to make explicit and then leverage these capabilities. These capabilities must take a central place in the language and not be abstracted away. In doing so, we ensure behaviors encoded in HLSR retain computational mechanisms important for intelligence.

There are four primary features unique and important to ISAs: 1) support for flexible autonomous behavior in complex, dynamic environments, 2) symbolic encoding of knowledge, 3) comparison and selection of alternatives via conflict resolution, and 4) pervasive and continuous adaptation through learning. These are each discussed in more detail below.

2.3.1.1 Blended Reactivity and Goal Directed Behavior

ISAs support both reactive and goal directed behavior. To achieve reactivity, ISAs support non linear control constructs such as productions, rapid context switching, and parallelism. At the same time, they also provide constructs and processes to enable rational, goal-oriented behavior. This balance is achieved in cognitive architectures with flexible goal representations, least commitment to execution path (section 2.3.3.1), and serial decisions.

ISAs support reactivity through interrupt driven behavior. Context switching (i.e., changing the computational focus) is important in both Soar and ACT-R. Soar supports this behavior with problem spaces and goal decomposition realized through the impasse mechanism. However, abstracting away from Soar, generally architectures "name" contexts and associate sets of knowledge with specific contexts. The ACT-R 5 buffer mechanisms support context switching at the architectural level.

To provide behavior consistency, ISAs must constrain reactive behavior. A common mechanism for this a decision making process. The decision process is used to direct behavior in a consistent and goal-oriented manner. Soar supports an explicit "perceive-decide-act" cycle, with "decide" corresponding to its most primitive deliberate act. All activities (e.g., knowledge retrievals) can occur in parallel within a single PDA loop, but the sequence of PDA loops forms a higher level serial structure. Similarly in ACT-R each module (e.g. declarative memory) operates sequentially, with the added constraint that they can only communicate with each other through constrained buffers (one chunk at a time) to the central production system. However, all modules operate independently in parallel, and within each module the operations are best understood as massively parallel (i.e. match all chunks or productions at the same time).

ISAs also include automatic processes for failure detection, and they integrate these processes with reactivity and goal driven behavior. Soar's impasse mechanism is a general failure detection process. Failure detection processes should enable improved robustness/more graceful degradation when facing novel situations because the system receives an explicit signal regarding the failure. Penetrability of the reasoning process including its symbolic structures (as described in 2.3.1.2) may also be important for robustness. Such penetrability is often referred to as meta-cognition. In practice, ISA support for meta cognition is relatively weak. To be

effective, the developer must provide additional knowledge structures to track the details of reasoning in order for robustness and graceful degradation of behavior to be achieved [5, 31].

2.3.1.2 Symbolic Encoding of Knowledge with Associative Retrieval

ISAs provide mechanisms for encoding symbolic representations of knowledge. Symbolic structures supported by Soar and/or ACT-R include: productions, goals, operators, chunks/WMEs, and preferences. ACT-R makes a distinction between symbolic, declarative knowledge (chunks) and procedural knowledge (productions). Soar does not make this distinction, as all long-term knowledge in Soar is represented as productions.

Knowledge can be either penetrable (i.e. inspectable from other knowledge) or *impenetrable* (i.e. not inspectable from other knowledge). In Soar, only declarative memory is fully penetrable. A subset of symbolic preferences is also penetrable. Similarly, in ACT-R only chunks are penetrable. However, penetrability is orthogonal to the symbolic/sub-symbolic distinctions. A symbolic structure can be impenetrable (productions in Soar and ACT-R) and sub-symbolic values (like activation) could be penetrable.

Retrieval of knowledge is strongly associative in ISAs. Associations can be explicitly represented in the architecture implementation (e.g. Rete algorithm production match [15]) or implicit (e.g. learned co-occurrences). Associative retrieval enables reactivity as it allows the existence of a pattern in memory to directly trigger a behavior. .

2.3.1.3 Comparison and Selection of Alternatives via Conflict Resolution

ISAs provide built-in mechanisms for weighing and choosing among alternatives. Alternatives to be weighed include the desired goal(s) (section 2.3.2.1), the action(s) that should be taken (section 2.3.2.3), and what should be believed given prior and current observations

(section 2.3.2.2). Conflicts typically arise because each architecture imposes specific constraints on representation. The form of this constraint is often a single "slot" or storage location. For example, Soar provides only a single operator slot in each state; therefore, a conflict resolution process must select between alternative operators to determine which gets to reside in that slot.

A conflict resolution procedure can be fixed or knowledge-mediated (deliberate). A fixed procedure simply activates some selection algorithm and can be executed immediately, without further reference to the context. A knowledge-mediated process relies on further knowledge to resolve the conflict and is thus very sensitive to the context. In general, beliefs are mediated by non-deliberate processes and goals by deliberate ones. Soar and ACT-R differ with respect to memory transformation. Soar operators require symbolic preference knowledge for selection while ACT-R productions rely on sub-symbolic processes for selection.

To aid with the conflict resolution process, ISAs contain structures that represent the relative or absolute weight of particular choices. We refer to these structures as preferences. Preference knowledge can be symbolic (as in Soar's operator evaluation rules) or sub-symbolic (as in ACT-R's production utilities). Preferences influence which alternatives are selected.

2.3.1.4 Pervasive, Continuous Adaptation through Learning

ISAs, and in particular cognitive architectures such as Soar and ACT-R, provide learning mechanisms that can tune and extend agent knowledge. ISAs provide mechanisms that control when learning occurs, what form learned knowledge takes, and when learned knowledge is applied.

Learning has often been a roadblock to development of higher level representations [51]. The primary difficulty imposed by learning is the inability to associate learned knowledge to the higher-level representation. We discuss this issue at length in section 4.1.3.

2.3.2 Common Structures in Intelligent Systems

ISAs share many structural elements among them, although the literature for each ISA uses its own terminology, so the similarities are often not obvious on the surface. Our analysis abstracts away from terminological differences and exposes similarities at different levels of details and with different parameters for each ISA. We find that many common structural elements have primary structural and functional purposes that are closely related. Here we discuss the most important of these structures for intelligent behavior and, specifically, the structures from which we defined the HLSR primitive structures.

2.3.2.1 Goals

A goal describes a desired state or outcome. For example, a goal may be "send a message" or "the target is destroyed." Though the term "goal" is used in different ways in ISA's, functionally it retains the same core purpose. It serves both to provide an objective that reasoning will attempt to achieve, and as a context for narrowing reasoning to the most relevant and important activities. An activated goal is sometimes referred to as an *intention*[3] in the literature, and represents a current objective.

In ACT-R, goals have an explicit declarative representation. ACT-R supports a latent (or uninstantiated) representation of goal in the form of goal chunk definitions that define the classes of goal an ACT-R model can have. A goal is *considered* when a production that creates that goal is activated by ACT-R's sub-symbolic production matching process. A goal is *committed* or

---

[3] However, the literature includes several definitions of *intention*, including an active goal, a selected plan, or a step in a plan. Our definition is equivalent to *active goal*.

"activated" when it is instantiated and placed in the goal buffer. An activated goal serves as the focal point for production firings; that is, most productions test the current goal as part of their firing conditions. A goal is returned to the considered state by the sub-symbolic production matching system any time a production that would replace the current goal is a candidate for firing. Once a goal is achieved, or some other goal needs to be processed, the goal is removed from the goal buffer and stored in long term declarative memory where it resides in a latent state.

Current versions of Soar define do not define an explicit declarative representation of a goal. However, goals play an important part in any Soar behavior model. Because Soar does not provide an explicit goal representation, the developer is free to design their own representation. Two representations have become common among Soar developers. The first is oriented toward a built-in stack structure within the Soar architecture, which pushes and pops problem-solving context placeholders in response to reasoning impasses [25]. Developers may use this stack of reasoning contexts to represent a hierarchy of goals and subgoals. In this particular representation, each goal is created without an initial latent state, instead being generated automatically in the active state when needed. However, the information regarding the goal, such as its name and relevant knowledge structure, is latent in Soar productions, which are Soar's only form of long-term memory. Using this approach, the architecture automatically activates new goals in response to reasoning impasses (the system does not have the knowledge to select a unique discrete action for its next reasoning step). An automatically activated goal becomes a sub-goal, which is used as a new context in which operators are proposed to perform internal or external actions. Automatic sub-goals may be returned to the considered state on a decision by decision basis, based on whether the impasse that caused the goal has been resolved.

If the impasse becomes resolved, the sub-goal is popped from the stack and becomes latent again. No record of this instantiation of the goal is kept in memory for the future.

The other common approach within Soar is to represent goals using the same representation language as all the other features describing a reasoning context. In this case, instead of relying on architectural goal-activation and representation, the developer creates user defined goal structures, similar to those created in ACT-R, and manages their relationships through Soar productions. There are many different approaches to managing these goals, and the goal state (latent, active, etc.) is defined and controlled by productions, as it is with beliefs (see 2.3.2.2).

In addition to the CCRU states, goals can typically also be marked as *achieved.* A goal is achieved when the objective defined by the goal is met. In most cases, a goal that is achieved is reconsidered and then moved back into the latent state. However, this does not always have to be the case. There is typically one or more logical condition that defines whether a goal is achieved. Sometimes the condition is as simple as a tag indicating that the goal has been completed, other times it is more complex. In both Soar and ACT-R these conditions are encoded in productions.

2.3.2.2 Beliefs

A belief represents either a perceived or inferred relationship among objects. These objects could be concrete as in the case of real-world objects like tanks and planes, or could be abstract as in the case of concepts such as "danger" or "action". An activated belief is said to be *believed* by the behavior model. Modification of beliefs is the primary result of ISA decision processes. Most behavior model processing is concerned with maintaining a useful and relevant set of beliefs.

In ACT-R, beliefs have an explicit declarative representation in the form of *chunks*. Each chunk is associated with a chunk-type, indicating the structure of the belief. Beliefs describe relationships between objects, and ACT-R supports this through labeled links to other chunks, and thus chunks can represent any arbitrary set of relationships. Latent beliefs are partially defined by chunk-type definitions and are instantiated by productions. The ACT-R representation of a considered belief is less clear and explicit. Beliefs are instantiated in buffers before being moved to long term memory. Furthermore, they are associated with sub-symbolic activation levels, which help determine whether they will be retrieved. A belief can be thought of as activated when it resides in long-term declarative memory and has sufficient activation to be retrieved for some purpose. A belief returns to the latent state when its activation is no longer sufficient for retrieval or when other conflicting beliefs are retrieved in its stead.

In Soar, beliefs are represented as collections of related working memory elements (WMEs), each of which is a triplet defining a symbol, a relation, and another symbol or value. Sets of WMEs form a graph structure that can, just as ACT-R chunks can, represent any arbitrary set of relationships. Latent beliefs are sometimes implicitly encoded in productions called "elaborations." Considered beliefs take the form of a preference to believe. Activated beliefs are instantiated in working memory. Soar does not have a long-term declarative memory, so there is no built-in representation for what was believed; that is, latent beliefs that were once activated.

2.3.2.3 Transforms

A transform represents an action to be carried out in an attempt to achieve one or more goals. Not every ISA has an explicit representation for a transform, but they always provide

some functional equivalent. An activated transform *executes*, generally by either changing the belief set or initiating some motor action.

ACT-R does not have an explicit representation for transforms. Instead, it has a set of loosely coupled productions that fire sequentially to achieve a goal. Because of this it is not possible to directly assign a CCRU states to transforms in ACT-R. However, if a single production is treated as a mini-transform, a mapping can be made. A latent transform in ACT-R then maps to a production in long-term declarative memory that is not currently activated. A considered transform is an activated production that the sub-symbolic production selection process is considering for selection. An activated transform is a production that is firing. For larger-scale collections of productions, ACT-R considers them on a production by production basis.

Soar explicitly defines an operator construct that directly implements the concept of a transform. The decision process in Soar is organized around proposing and selecting operators for application. A latent transform then exists in the form of a production that proposes an operator. A considered operator is an operator that has been proposed but not selected. An activated operator is one that has been selected for application. After application an operator is automatically reconsidered and typically is removed from memory, thus it is not retained in an explicit latent state.

In addition to the CCRU states, transforms are associated with additional states related to execution. Often, behavior models track these states in order to appropriately sequence behavior. We summarize these states as follows:

- *waiting*: activated but not yet executing
- *executing*: activated and executing any transformations it defines

- *succeeded*: completed and successfully executed all intended transformations
- *failed*: completed but without executing all intended transformations
- *suspended*: started executing but stopped because of interruption

The Soar and ACT-R architectures provide some built-in mechanisms for detecting these states, but those related to success and failure must typically be defined by developers when they are required for a particular behavior.

2.3.2.4 Preferences

A preference represents a choice or a hint regarding a decision. Decisions change the state of object from considered to either latent or activated. While decisions within ISAs are always made by a built in decision process, often preferences influence decisions in the direction that best supports any active goals. An activated preference is one that applies, that is, it is being used to help make the decision

In ACT-R, the built-in preference mechanism is sub-symbolic. That is, preferences are not present in declarative memory, but instead are numeric and managed by the architecture and by statistical parameters such as base-level activation, production costs, and probabilities of success and failure. Latent preferences are elements of the sub-symbolic values that are not currently being used for the selection process. There is no explicit representation of considered preferences, as sub-symbolic preferences are either used or not used based on a fixed process. However, activated preferences are used in activation calculations for selecting memory elements or productions.

Soar has a strongly knowledge-driven decision process and thus includes an explicit preference construct; however, it has been constrained in the current version of Soar to apply mainly to operators. Preferences for beliefs are allowed, but play a limited role in decision

making. Latent preferences exist as elements of productions, just like all long-term memory artifacts in Soar. Considered preferences exist internally within the architecture and are not available for inspection. Soar defines a preference hierarchy and conflict resolution process that determines which preferences are activated. An activated preference influences the activation of an operator, belief, or developer-defined goal.

2.3.3 Common Processes in Intelligent Systems

In addition to the structures described above, ISAs share common processes that serve as the basis for intelligent behavior. We summarize these processes below and discuss their impact on HLSR's design.

2.3.3.1 The Decision Process and Least Commitment to Execution Path

ISAs conform to the **principle of least commitment to execution path**; that is, they delay decisions until the last moment when it is necessary to make the decision. This is in sharp contrast to traditional software systems where a large number of decisions are made prior to compilation of the program, and decisions that are left to runtime are highly constrained by traditional control logic.

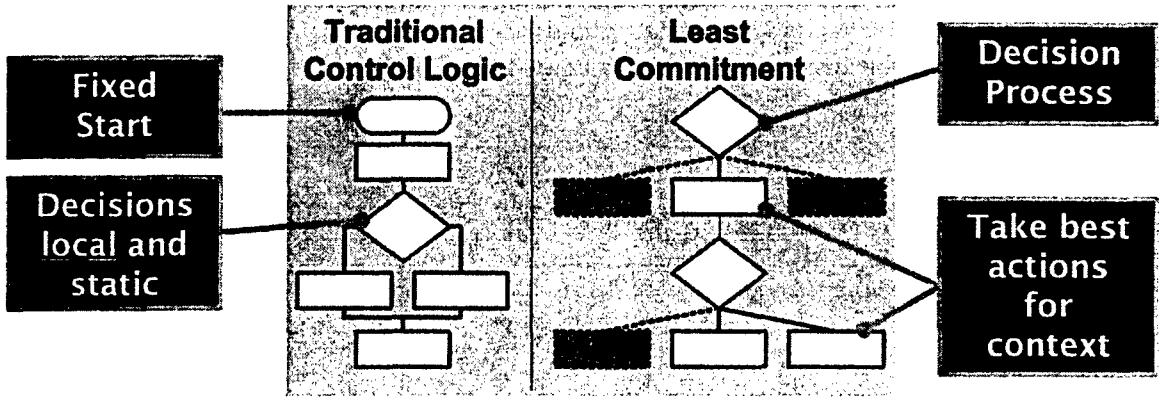


Figure 5: Least commitment contrasted with traditional control logic.

To support this principle, each ISA defines a decision process. The decision process runs continuously and attempts to bring the most relevant knowledge to bear on each decision step. This decision process is diagrammed at an abstract level in Figure 5. The decision cycle is sometimes also referred to as the "decide-act" cycle because it involves the selection from a set of alternative actions and then carrying out the selected actions.

HLSR supports the ISA's decision process and least commitment to execution path in the following ways:

- HLSR does not provide fixed sequential control logic.
- HLSR does not provide a decision mechanism, that is, no code written in HLSR can force a particular action. HLSR code only provides hints and preferences.
- HLSR does not allow transforms to invoke other transforms; that is, no process can begin execution without a decision being made first.

These constraints and design decisions enable the compiler to leverage the unique decision capabilities of the underlying ISA as discussed in 2.3.1.

2.3.3.2 CCRU

Each ISA representational structure can be described as existing in one of three states: latent, considered, and activated. This abstraction can be extended further to include the transformations that change the state of an ISA structure. These transformations are as follows:

- Consider: Changes the state of a structure from latent to considered. Can be thought of as the process of initially bringing a structure up for conflict analysis.
- Commit: Changes the state of a structure from considered to activated. Can be thought of as the process of making a decision among a set of possibly conflicting options during conflict analysis.

- **R**econsider: Changes the state of a structure from committed to considered. Can be thought of as the process of re-evaluating a structure to see if it still is appropriate for the current context.
- **U**nconsider: Changes the state of a structure from considered to latent. Can be thought of as the rejection of a structure for the current context.

These processes, referred to as the CCRU processes, are abstractions of the sub-processes involved in decision making; therefore, they directly support the principle of least commitment to execution path discussed in 2.3.3.1. Figure 6 shows the CCRU processes and how they relate to the CCRU states.

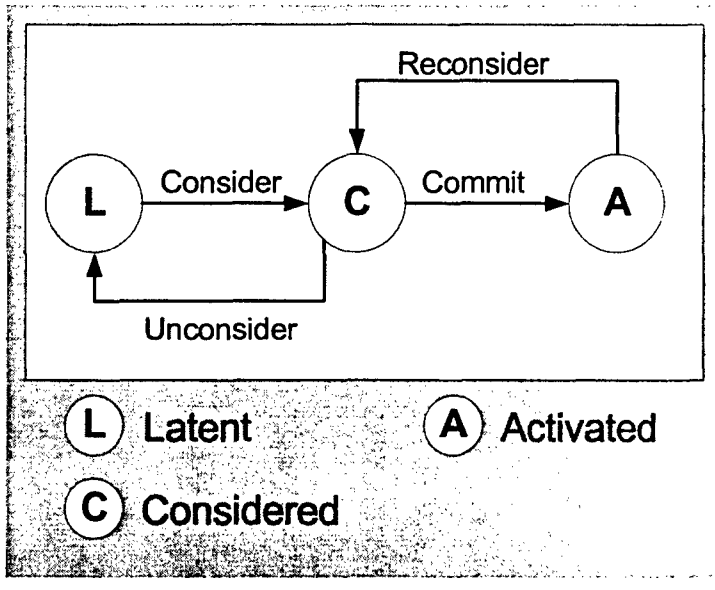


Figure 6: Consideration, Commitment, Reconsideration, Unconsideration and the three CCRU states.

The CCRU processes have been abstracted and formed into primitive memory operations in HLSR. CCRU processes are good candidates for such primitives because they map well to the key principles of intelligent behavior and because they are reflected in the underlying ISA. Because HLSR is at a higher level than ISAs, the CCRU states and processes in HLSR apply only at the symbolic level. However, these processes map during compilation to one or more

CCRU processes and states in the underlying architecture. Section 2.5.2 provides more details on these primitive HLSR operations.

2.3.3.3 *Goal Driven Behavior*

All ISAs exhibit goal driven behavior. Goal driven behavior is the mechanism by which ISA's balance reactivity and least commitment to execution path with behavior consistency. In HLSR, goal driven behavior is implemented as a high-level combination of CCRU processes. The core constructs are goals (see 2.3.2.1) and transforms (see 0). Figure 7 shows how goals and transforms interact to form goal driven behavior.

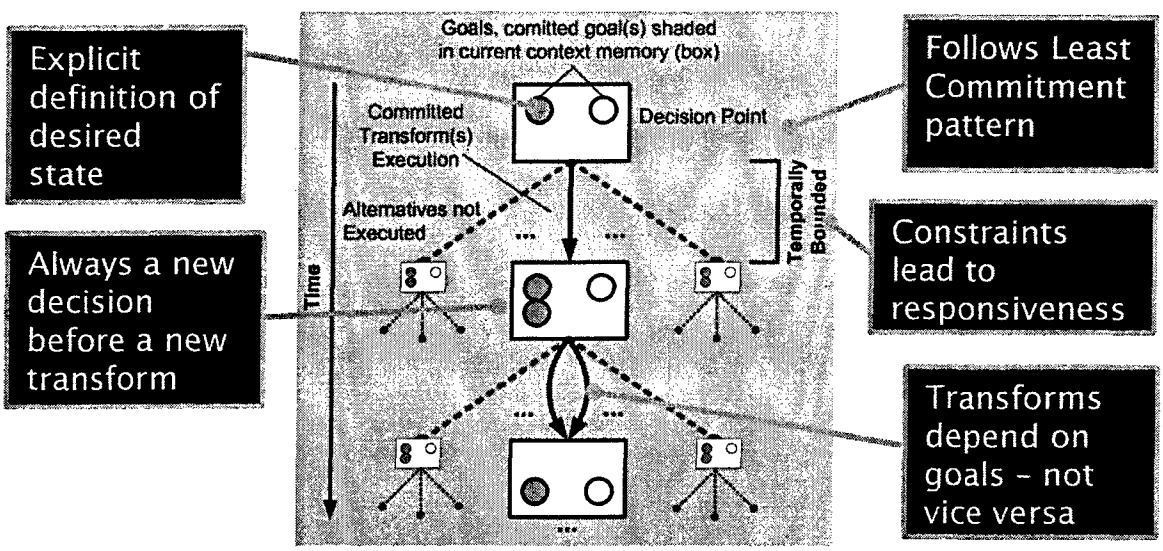


Figure 7: Goal driven behavior.

The behavior pattern consists of four steps as follows:

1. Consider one or more goals
2. Commit to some subset (possibly all) of the considered goals (activate them)
3. Consider one or more transforms to achieve the activated goals
4. Commit to one or more transforms to achieve the activated goals

This process repeats to form a goal-directed decision process. Architectures vary in how they constrain this process. In Soar, only one transform may be activated at any one time. In ACT-R, only one goal may be activated at one time. In Soar, using the stack structure for goals, a stack of goals and transforms may be active at one time; however, dependencies between the goals and transforms in the stack usually lead to only one transform actually executing at one time. These constraints reflect the two architectures' emphases on psychological plausibility. Architectures with a more functional emphasis sometimes allow multiple simultaneous active goals and transforms in arbitrary configurations.

Goal driven behavior narrows the context for decision making. This is critically important for knowledge rich systems and differentiates ISA-based models from stimulus response-based models. Only transforms that apply to activated goals are available for consideration, thus achieving a narrowing of focus that allows a behavior model to execute complex tasks that involve significant reasoning.

Goal driven behavior serves as the core high-level process within HLSR. It is around this process that HLSR programs are organized. HLSR does not define the details of the process (e.g. such as sequential v. parallel goals), but does provide constructs and constraints that emphasize and maintain the characteristics of goal driven behavior. These constructs include goals and transforms discussed in section 2.5.1.2. The constraints are as follows:

1. Only goals and transforms can be reactively considered. This constraint is reflected in ISAs and helps balance reactivity and behavior consistency by requiring that all significant context changes go through the goal driven behavior process.
2. Transforms cannot be considered without the context of an activated goal.
3. A transform instance can only be considered for a single goal.

These constraints provide a framework within which intelligent behavior can be encoded. This framework implies that to execute a behavior, the four steps above should be followed. A goal should be created that represents the desired outcome of the process. One or more transforms should be constructed that help achieve the goal. In addition, preferences should be specified to guide the decision making process. These constraints form the primary execution model for intelligent behavior much like the fetch-execute cycle and sub-routine calls form the foundations for traditional software execution.

2.3.3.4 Reactive Consideration

ISAs combine goal driven behavior with reactivity. All ISAs have built in mechanisms for reactive consideration. By reactive we mean that the consideration is the result of some internal or external stimulus rather than a purely algorithmic process.

A critical feature of ISAs is that they allow very few reactive commitments. Commitments should be made after conflict resolution, not before. This constraint is required in order to meet the principle of least commitment to execution path. If reactive commitments are allowed, behavior is essentially hard-coded, not taking into account the broader context, and inappropriate decisions might be made in the following ways:

1. Behavior can be wrong for the given context
2. Behavior can be inconsistent and poorly focused

As discussed in section 0, goal driven behavior processes are employed to alleviate these problems. Given these constraints, ISAs tend to define two paths to reactive consideration, with a third path being a pseudo-reactive path. First, most ISA's allow the definition of patterns, usually in the form of rules, that consider some construct when they match. Second, ISA's almost always allow external events to be injected into the behavior model through some sensory

system. Third, the architecture may have internal mechanisms that bring up constructs for consideration as a side effect of processing. An example is ACT-R's spreading activation, which can retrieve elements for processing that were not specifically requested. We refer to this third path as a "pseudo-reactive" path because it is a mixed result of deliberate processing (e.g., a retrieval request) and architecture driven stimulus (e.g., spreading activation).

ACT-R support all three of these mechanisms, however, the first mechanism is used more rarely in Soar. That is, ACT-R models tend to be more goal driven than Soar models in general. Soar supports the first two, with the additional ability to create reactive commitments to beliefs. However, this practice is considered to be poor programming practice in most cases.

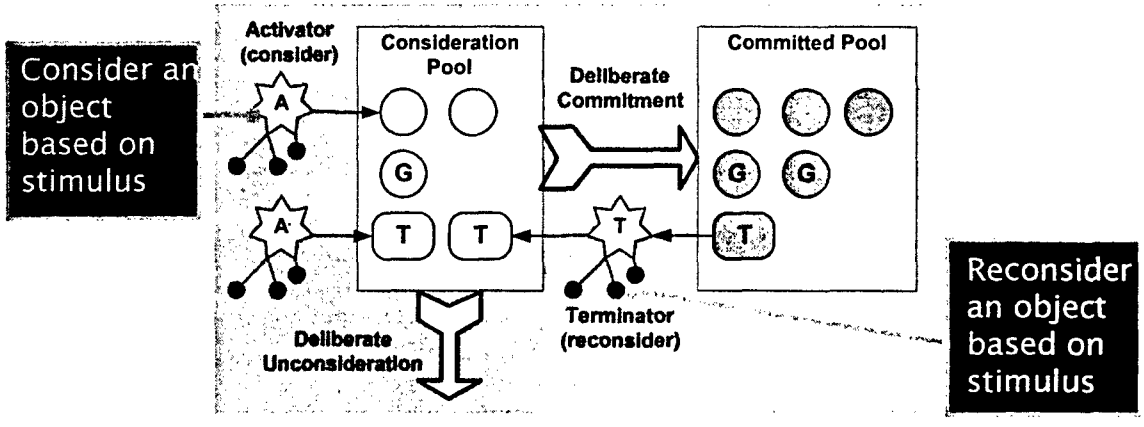


Figure 8: Reactive consideration of memory constructs.

HLSR provides primitive constructs to enable reactive consideration (Figure 8) at the symbolic level. These constructs are activators and terminators, and they consider and reconsider objects respectively based on a pattern (see Section 2.5.1.1). An important constraint is that both activators and terminators can only consider or reconsider goals and transforms. Beliefs can only be considered based on deliberation; that is, by transforms within the goal driven process. This constraint is based on the ISA constraints discussed above and helps prevent the two behavior issues discussed above.

2.3.4 The Principle of Architecture Discretion

To this point, we have primarily discussed the common themes in ISAs. At a conceptual level and, to some extent a design level, ISAs appear very similar. This is moderately surprising given the fundamental implementation and functional differences of these architectures. The challenge of HLSR is to take the unifying concepts between ISAs and somehow map them to significantly different implementation constructs and constraints, while retaining sufficient developer control to make the make the engineering task tractable.

The mechanism by which HLSR makes this mapping possible is the **principle of architecture discretion**. The principle of architecture discretion says "if a construct or constraint is not defined at the HLSR level, then it is up to the architecture and/or compiler to define it." That is, the architecture is given significant discretion as to how to execute HLSR knowledge. Figure 9 shows how architecture discretion affects the relationship between HLSR and ISAs.

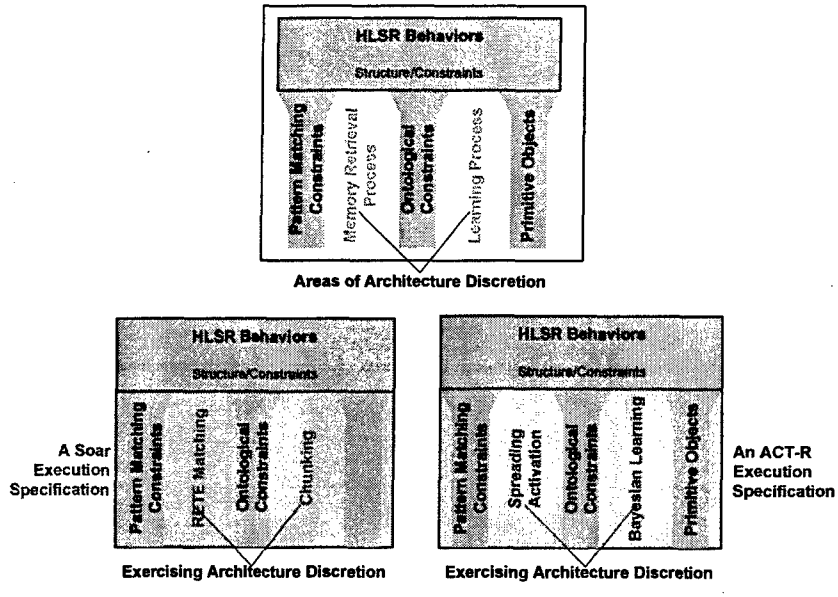


Figure 9: Exercising architecture discretion.

HLSR provides a set of constructs and constraints that partially specify behavior. Within this framework, the compiler and architecture is given discretion to define constructs, constraints, and processes specific to that architecture. We call these constructs, constraints, and processes not defined by HLSR *areas of* discretion. The example shows two such areas of discretion: memory retrieval process and learning process. The orange and green figures show how these areas would likely be implemented in Soar and ACT-R respectively.

Table 3 describes the specific areas of discretion defined by HLSR. It is apparent that this list includes a significant number of critical constructs, constraints, and processes for behavior execution. The result is that HLSR code by itself is not sufficient for execution. Because of this, we say HLSR is a *constraint language* rather than a programming language. That is, HLSR defines constraints within which the compiler and architecture produce behavior appropriate for a particular architecture.

Area of Discretion	Description
Memory structure	The number and form of memory partitions (e.g. long-term v. short-term memory), the primitive structures used to store individual elements, the size of any memory partition or memory element, and the size and form of any primitive data structures (e.g. strings, or numbers)
Object structure	The size, format and layout of HLSR objects in architecture memory.
Retrieval process	The process used to retrieve HLSR objects and relations from memory including whether and when *retrieval failures* and partial *retrievals* occur.
Retrieval strategy	When retrieval is attempted, what elements are retrieved, and the order in which elements are retrieved.
Default commit/unconsider process	The process used to commit/unconsider objects when insufficient HLSR knowledge is available to guide the commitment process.
Decision Process	The process constraining knowledge execution. This includes but is not limited to decision cycles, constraints on parallel execution, and architecture-specific mechanisms for resolving conflicts.
Error handling process	The process for identifying failure conditions and processing the failure information, including construction of HLSR failure objects.

Learning process	The architecture-specific learning process(es) and the form of architecture-specific learned knowledge.
Sensory System	The structure of the sensory system, the underlying data structure of sensory data, the timing and synchronization of sensory information with the external world.
Motor System	The structure of the motor system, the underlying format of motor system commands, the procedure executed by a motor command, the timing and synchronization of the motor system with the external world.

Table 3: List of areas of architecture discretion.

To be executable, HLSR requires a combination of HLSR knowledge and compiler/architecture defined constructs, constraints, and processes defining the areas of architecture discretion. These compiler/architecture defined constructs, constraints, and processes together form an *execution specification*. HLSR knowledge can be executed given an execution specification. HLSR provides very few constraints on how the execution specification defines the areas of discretion; therefore, it is possible, and even likely, for multiple execution specifications to be defined for a single ISA.

The impact of architecture discretion is that the behavior of HLSR models will vary based on the ISA and even the compiler. Specifically, the same HLSR model will not produce the same behavior when executed on two different ISAs or compiled with two different compilers for the same ISA. To see how behavior may vary, we present Figure 10.

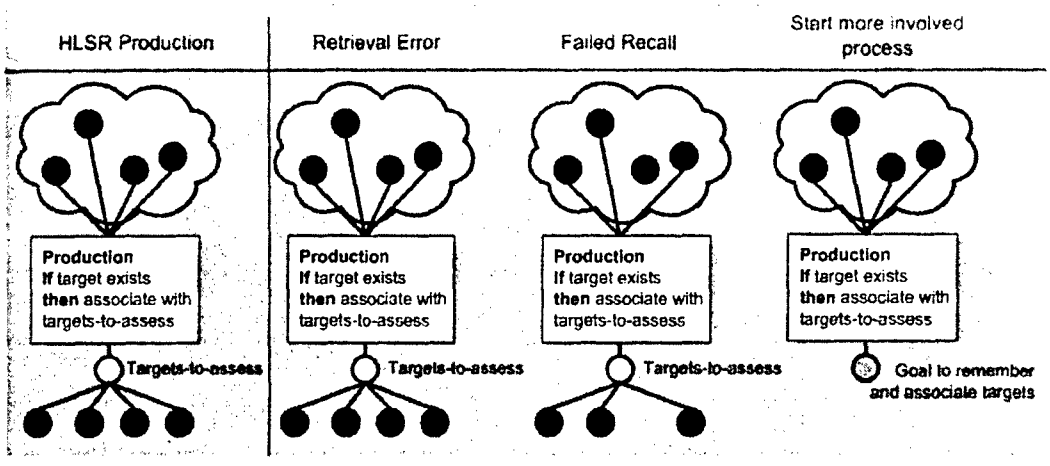


Figure 10: Example of how behavior can differ given different ISAs and compilers.

In Figure 10 we show a single production in HLSR (represented abstractly for compactness). It does a conceptually simple process; it links all targets in memory to a structure representing the group of targets called "targets-to-assess." In English, it says "I want to think about all targets collectively as a group." To the right of the vertical dividing line, we see three different ways of executing this production in an ISA. In the column labeled "Retrieval Error" the architecture retrieved the wrong object (an object that is not a target) and associated it with targets. In the column labeled "Failed Recall" the architecture fails to retrieve one of the targets from memory, and thus cannot associate it with the group "targets-to-asses." In the column labeled "Start More Involved Process" the architecture does not associate all targets in one step; rather, it creates a goal to associate the targets in multiple steps. Clearly the end behavior is different for each of the execution processes. What is invariant is the constraints provided by the

HLSR production; in all instances the architecture attempted to execute the reactive association of targets to "targets-to-asses."

In addition to allowing HLSR to map successfully to different ISA's, architecture discretion allows us to achieve two related goals. First, it provides the compiler with enough flexibility to leverage the underlying architecture; that is, take advantage of the special capabilities implemented within a particular ISA. Second, it enables HLSR behaviors to execute with the unique characteristics or "feel" of the underlying architecture. The cost, of course, is behavior predictability between ISAs and compilers as discussed above.

2.4 HLSR Requirements

Analysis of the requirements of developers and underlying ISAs leads to the definition of the core requirements for HLSR. We have defined six core requirements for HLSR to guide the analysis and design process. Our initial efforts focus primarily on making HLSR general enough to generalize concepts shared by Soar and ACT-R. We anticipate that future work on HLSR will include the core concepts from additional architectures, and thus the requirements will be generalized appropriately. The HLSR core requirements are as follows:

1. **HLSR must be independent of the target implementation architectures.** HLSR must not depend on the existence of particular individual architectural capabilities and structures that are not generally shared by other architecture.
2. **HLSR must be a high-level language,** similar to high-level programming languages. That is, an HLSR developer must not be required to code any low-level knowledge to produce an executable model. Our current working definition of "low-level" includes constructs that could clearly be implemented in alternate ways (by different underlying architectures); as well as constructs that do not

immediately provide a knowledge-level understanding of the model's behavior. Forcing HLSR to be a high-level languages is intended to make producing a behavior more efficient (more time spent on the desired behavior and less on details) while at the same time making the models in HLSR more maintainable.

3. **HLSR must make it both possible and convenient to package knowledge** into knowledge components. A component-oriented approach should allow developers to construct libraries of reusable behavior components, and definite implementation-independent interfaces for those components.

4. **HLSR must guide and make it easy for the developer to take advantage of the specific features of cognitive architectures**. It is not sufficient merely to provide a high-level computational programming language for knowledge-intensive agents. Rather, it is necessary to include those components and processes that have been identified in the research on cognitive architectures as being critical for intelligent and autonomous behavior.

5. **HLSR must support incremental addition of knowledge** efficiently and robustly. Model developers must be enabled to improve the knowledge-base of particular behavior models over time, without having to refactor significant portions of the initial knowledge base. Incremental addition f knowledge should also facilitate automated agent learning.

6. **HLSR must be complete and transparent**. HLSR must be expressive enough that HLSR models can be compiled and executed on each supported architecture without the need for additional architecture-specific code

These core requirements can be organized into three categories related to the developer, reuse, and architecture, as shown in Figure 11.

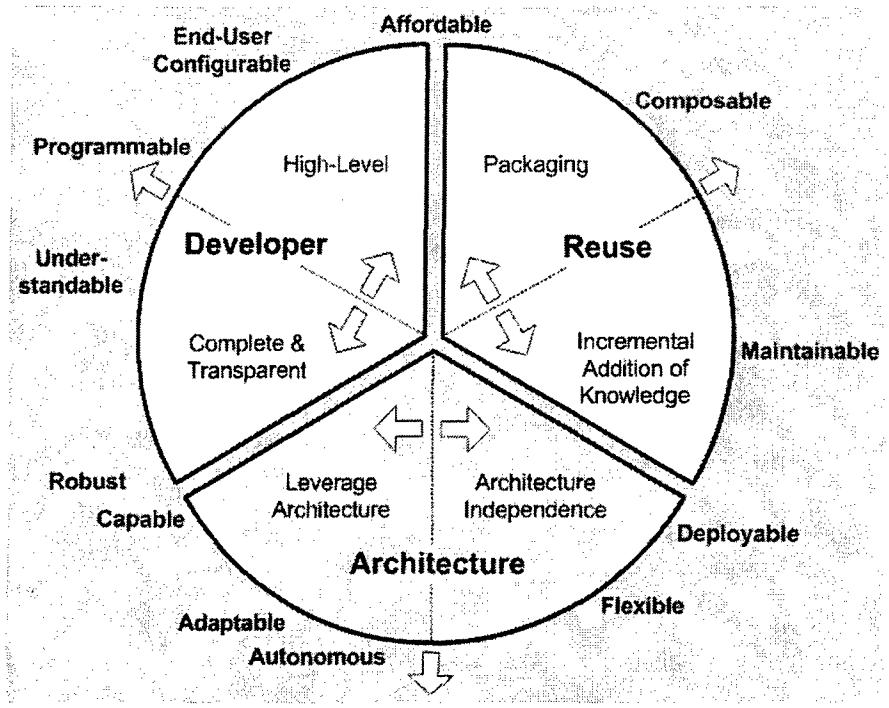


Figure 11: HLSR requirements compete.

The outside ring of blue represents customer requirements, or requirements of applications that use behavior models. These customer requirements roughly map to the functional requirements closest to them in the requirements pie.

Organizing the requirements this way clearly shows HLSR is pulled in six distinct directions, as all six requirements compete with each other. For example, leveraging cognitive architectures (requirement 4) necessarily involves drawing the developer's attention away from high-level behavior and toward architecture details. Furthermore, it leads to highly complex and unmaintainable interdependencies between knowledge. These interdependencies are necessary for context driven behavior. Thus, leveraging the architecture competes with reuse requirements. Even within the architecture slice, leveraging the architecture competes with architecture

independence in that maximally leveraging an architecture requires intimate knowledge and control of the architectures features.

This competition means that HLSR cannot possibly fulfill any of the requirements completely. Rather, our approach in HLSR is to balance the core requirements, as shown in Figure 12, thus creating a language that will be useful in general for intelligent system implementation.

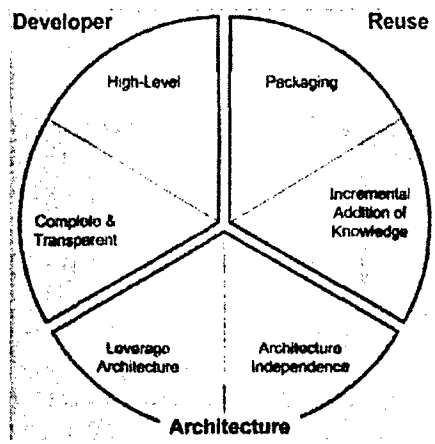


Figure 12: Balancing the requirements of HLSR. The colored lines indicate the extent to which HLSR should meet each requirement.

This goal to balance the requirements separates HLSR from many of the other attempts to provide tools for developing intelligent systems. For example, approaches that are based on traditional object-oriented techniques strongly concentrate on decomposition and reuse concerns, but have left the details of implementing high-level constructs and intelligent behavior processes to the developer. Other systems [51] have focused on making the specification of intelligent behavior higher-level and easier, but sacrifice the developer control necessary to generalize solutions to other problems and domains. Cognitive architectures and their associated languages

have provided very powerful core capabilities for intelligent behavior, at the cost of increasing the complexity and interdependencies among behavior components.

2.5 Formal HLSR Specification

HLSR is described in *Specification of a High Level Symbolic Representation (HLSR) for Intelligent Systems*. This specification includes a detailed description of every HLSR construct and constraint as well as a language grammar. Here we summarize the HLSR specification and point out important processes and constructs. Emphasis here is placed on the reasoning that led to each construct and constraint.

2.5.1 Core Primitive Constructs

The HLSR specification is primarily a definition of HLSR's primitive constructs. Because HLSR is a high-level representation, these constructs are necessarily more abstract and constrained than the primitives of the low-level ISAs. However, some primitives have fairly direct relationships with primitives in the underlying architecture.

It is important for HLSR to define constructs that bridge the gap between behavior design and implementation on an ISA. This means that HLSR needs to include both constructs that map well both to design artifacts and constructs that map well to ISA primitives, thus allowing behavior models to be modified at the appropriate level of granularity.

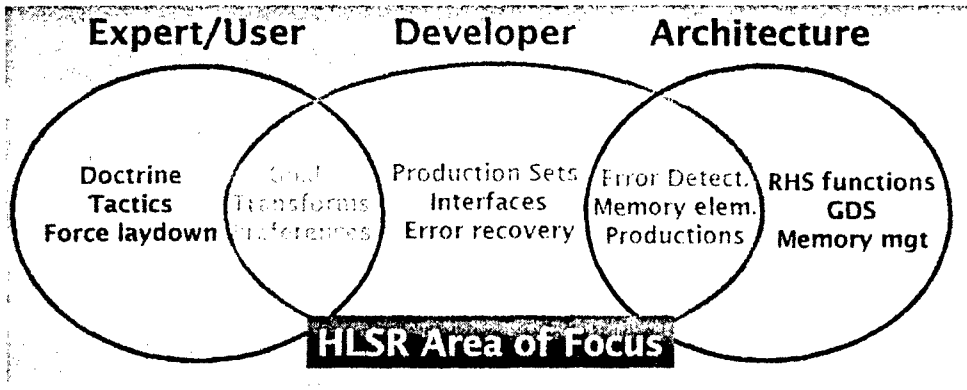


Figure 13: HLSR bridges gap between design and architecture.

Figure 13 shows how HLSR bridges the gap between design and ISA. On the left are constructs interesting to an expert or user. There is an overlap of some of these constructs with engineering design constructs to aid an engineer in encoding a solution to the expert/user's problem. Other design constructs are only useful to the developer. These are shown in the middle of the white oval. Typically these constructs are those that help an engineer structure and maintain a behavior model, such as interfaces and encapsulation mechanisms. Finally, there are constructs relevant to architecture designs that are used for managing execution. These constructs overlap with the constructs that developers require in areas where the developer needs to control architecture processes. This division is not perfect. Clearly some constructs, such as goals, are important across all levels. However, HLSR versions of these constructs are more abstract and highly structured than the architectural counterparts, and thus map better to design constructs than to ISA primitives.

Table 4 lists all of the major constructs defined in the HLSR specification. They are sub-divided into categories based on their primary use within a behavior model. Each construct has a form (middle column), which is used to express and encapsulate that construct in HLSR.

Construct	Form	Responsibility
Reactivity		
Activator	Production (rule)	Reactive consideration of goals and transforms
Elaboration	Production (rule)	Reactive association of beliefs with an activated transform
Terminator	Production (rule)	Reactive reconsideration of goals and transforms
Goal Driven Behavior		
Goal	Object w. knowledge about achievement	Explicitly represent of a desired state or objective
Preference	Production (rule)	Context based hints/choice
Transform	Object w. execution body	Manipulate beliefs and initiate external commands.
Encapsulation and Packaging		
Object	Object	Represents belief in memory
Interface	Collection of queries and manipulators	Decouple objects and procedures
Manipulator	Named transform fragment	Encapsulate related belief manipulations in a reusable construct.
Production Set	Object w. productions	Shared context for productions
Query	Named LHS condition	Encapsulate memory pattern in a reusable construct

Table 4: HLSR primitive constructs.

The subsections below discuss briefly each of these constructs by category. Further details are available in *Specification of a High Level Symbolic Representation (HLSR) for Intelligent Systems*.

2.5.1.1 Constructs for Reactivity

HLSR provides two constructs for symbol-based reactivity. By "symbol-based" we mean that these constructs specify patterns of symbols that define the reaction context (i.e. the pattern that triggers the reaction). Traditionally such constructs take the form of productions (or rules,

i.e. if X then Y) and this is also the case in HLSR. However, HLSR productions are constrained considerably more than production in other production-based systems.

HLSR provides three structures for reactive consideration, each with slightly different constraints. First, an **activator** is a production rule that can *consider* (section 2.3.3.2) one and only one goal or transform for activation. Second, a **terminator** is a production rule that can *reconsider* (section 2.3.3.2) one and only one activated goal or transform. Third, an **elaboration** is a production rule that can consider (or reconsider) one and only one belief for association with a transform's local memory (see 2.5.1.2). Activators and terminators allow engineers to specify conditions for behavior changes based on symbolic patterns that are sensed by the behavior model. Elaborations serve a more developer-oriented role by providing a mechanism for conveniently associating the proper memory elements with a transform, much like parameter lists do for functions in traditional software systems.

HLSR further constrains its productions by limiting what can appear in the production's memory pattern. Nested patterns (patterns within patterns) and universal quantifiers are currently not allowed. The functional capabilities these provide can be realized through queries (section 2.5.1.3) and iterative loops (*Specification of a High Level Symbolic Representation (HLSR) for Intelligent Systems*) respectively.

The reason for these constraints is mainly to make behavior models more maintainable and understandable. In doing so, these constraints also help prevent over commitment through excessive detail (see 2.2.2.1), and by forcing details to be specified in modular units (queries, see 2.5.1.3). Patterns that contain complex logic, like that prohibited by HLSR, are difficult to design and understand. Experienced knowledge engineers know that it is often better to break a

complex rule into several smaller, more modular rules. HLSR makes this a requirement and provides explicit support for this process.

2.5.1.2 Constructs for Goal Driven Behavior

As discussed in earlier sections, HLSR provides direct support for goal driven behavior. This support includes three constructs together with constraints on how those constructs are processed.

The first construct is the **goal**. HLSR goals are defined at a higher level than ISA goals, but map fairly directly to a combination ISA goals and some ISA infrastructure code. An HLSR goal is an object (see 2.5.1.3) implying that it can contain relations and be associated with object-oriented interfaces. An HLSR goal is always associated with the IGoal interface. This interface provides to two important queries – IsMet and HasFailed. The developer defines these queries (indirectly) for each type of goal. These queries indicate to the architecture and other HLSR knowledge when the goal is met and when the goal has failed (can never be met).

HLSR supports *achievement goals* and *maintenance goals*. An achievement goal is automatically reconsidered by the architecture whenever the goal becomes met or fails. A maintenance goal is only reconsidered automatically when it fails.

The second construct is the **transform**. The transform is the most complex of HLSR's constructs, and maps to multiple ISA constructs and processes. However, a simple transform maps fairly directly to the primitive forms of transforms provided in ISAs (section 2.3.2.3).

A transform is an object, as a goal is. It contains a *transform goal*, a set of relations referred to as *transform local memory*, a set of productions called *elaborations*, and a *body*. A transform goal is the goal that the transform will attempt to achieve. A single transform instance can only ever be associated with a single goal instance. Transform local memory is similar to

local function memory in traditional programming. It maintains the important relations relevant to transform body execution. Elaborations are discussed in 2.5.1.1. The transform body is the centerpiece of a transform. It contains a set of memory patterns, memory manipulations, and execution constraints that define a set of related processes that are required to achieve the transform goal.

A transform is also associated with the ITransform interface. This interface provides access to the transform execution states (see 2.3.2.3). The architecture executes two automatic processes on transforms. First, the architecture automatically reconsiders transforms under the following conditions: the transform succeeded, the transform failed, or the transform goal was reconsidered or unconsidered. Second, the architecture automatically tags the transform goal with the transform. This reduces the low-level code that must be written to manage the processing of goals (see 2.2.1).

The third construct is the **preference**. A preference is a production that instantiates a **preference hint** based on a memory pattern. Preferences are defined at a higher level in HLSR than in ISAs. In fact, ISAs do not always have built in support for symbolic preferences, but such support can be supplied through the compilation process and runtime libraries (see 2.6.2.3).

A preference reactively generates hints to the architecture about which objects, goals, and transforms should be committed or unconsidered at any given time. HLSR provides several types of preference hints with differing levels of specificity as to what decisions are preferable. First, unary priority hints assign absolute qualitative priorities to objects. An example is assigning the qualitative value "critical" to a goal. Second, a binary preference hint assigns a relative priority between two objects. An example is specifying that a goal to avoid a missile is more important than a goal to refuel. Finally, a unary selection hint specifies a preferred CCRU

process to execute on the object. This CCRU process can be either *commit* or *unconsider*. An example is specifying that a goal to refuel should be committed (changed to the *active* state).

The ISA is ultimately responsible for all decisions, and thus it is not required to execute preference hints. The ISA must apply its conflict resolution mechanisms together with any compiler generated preference hint management code to make final decisions as to the state of any considered object.

2.5.1.3 Constructs for Encapsulation and Packaging

HLSR provides several constructs that have the exclusive purpose of helping create units of behavior that can be encapsulated and reused. To some extent all HLSR constructs share a common theme of encapsulation. Productions, goals, and transforms all package and encapsulate important elements and processes needed for intelligent behavior. However, some HLSR constructs are designed primarily for this purpose.

The first of these constructs is an **object**, where here "object" is used in the object-oriented (OO) sense. An object is a collection of relations between one symbol and another. Objects can represent real-world objects (e.g. a tank) or concepts in memory (e.g. a task). In ISA terms, an object represents a **belief**. Two important characteristics of HLSR objects are as follows: all HLSR objects are associated with type definitions, and all HLSR objects are inherited (OO terminology) with the base type **symbol**. That is, an HLSR behavior model represents declarative memory as a set of symbols, just as ISAs do.

Type constraints are somewhat controversial among behavior developers. Developers with a scientific or psychological background tend to dislike object typing because they feel it reduces the runtime flexibility and psychological plausibility of the model. Developers with a

software engineering background tend to like object typing because it reduces data entry errors and increases modularity.

It is not object typing itself that is the limiting factor for runtime flexibility as much as how object typing has been implemented in OO systems. In OO systems types are specified by the developer within the code and cannot be changed. Furthermore an object can only be associated with one type during its lifetime. HLSR uses a more flexible typing system where types can be modified during execution, and an object's type can be changed during execution. This allows for more flexible execution while retaining the primary benefits of typing.

HLSR provides two constructs for encapsulating closely related memory patterns and memory manipulations. These are the **query** and the **manipulator**, which are close cousins of Prolog predicates and traditional software functions respectively.

A query applies a name and a parameter list to a collection (block) of logical memory pattern conditions. A query can then be invoked (executed) in places where the logical memory pattern encapsulated by the query is required. Queries are used to encapsulate parts of productions and transform bodies (see 2.5.1.2).

A manipulator applies a name and parameter list to a collection of memory manipulation statements. More specifically, a manipulator encapsulates a portion of a transform body. A manipulator can be invoked (executed) within a transform body or within another manipulator.

Queries and manipulators are important concepts because they provide a level of indirection that is important for achieving least commitment to dependencies (see 2.2.2.1). Rather than having a transform body or production depend on the details of some memory structure, they can depend on a higher level concept and set of parameters. The mapping to

lower level memory structure can be made in an isolated, central location that is easy to understand, modify, and maintain.

Queries and interfaces can be collected together to form **interfaces**. An interface is a named set of queries and manipulators. HLSR supports two types of interfaces. The first is an object-oriented interface, which, as its name implies, is associated with an object. Its queries and manipulators are used to access the object's relations. The second is referred to as a procedural interface and is closely related to callback interfaces in traditional software. A procedural interface is used by a transform to access knowledge that is necessary for transform execution but can vary depending on how memory is structured. In either case, interfaces are a key component for packaging and reuse because they explicitly define and encapsulate the knowledge elements that form the boundaries between different behaviors.

Production sets are the last of the HLSR encapsulation constructs. A production set is an object that encapsulates a collection of related productions. Productions within a production set can only fire when that production set is activated. Production sets also hold relations just as objects can. These relations form a common context shared by all productions, and the productions within a preference set can reference these relations in their memory patterns.

2.5.2 Primitive Memory Processes: CCRU

HLSR leverages the CCRU concepts described in 2.3.3.2 to form primitive memory operations. HLSR's CCRU primitives are at a higher level than ISA CCRU processes. Specifically, HLSR CCRU primitives apply to symbolic knowledge only, and are applied uniformly across all memory objects.

Using CCRU as the primitive foundation for HLSR memory operations leads to behavior models that emphasize least commitment principles. Furthermore, though HLSR CCRU

processes do not always map directly to ISA processes, they follow a similar pattern as ISA CCRU processes and thus tend to have natural mappings to ISA processes.

HLSR directly supports the four CCRU operations. They are implemented in a straightforward way as four language keywords:

- **Consider**: *consider* changes an object's state from latent to considered. The consider transform can execute on two classes of objects: existing objects in the latent state, and objects that are newly created.
- **Commit**: The *commit* transformation changes an object's state from considered to activated. Any special activation process associated with the object or relation may begin any time after the commit transformation is applied to it and before the reconsider transformation is applied to it. Only objects that are already created and are currently in the considered state may be committed.
- **Reconsider**: *reconsider* changes an object or relation's state from activated to considered. Any special activation processes associated with the object or relation must cease after the reconsider transformation is applied to it. Only objects that are already created and are currently in the activated state may be reconsidered.
- **Unconsider**: *unconsider* changes an object' state from considered to latent. Only objects that are already created and are currently in the considered state may be unconsidered.

In addition to the CCRU state transformation processes, HLSR provides several keywords for testing the CCRU state of an object. They are as follows:

- **Latent**: A *latent* object is an object that has been *unconsidered* or has never existed in memory. There are three testable sub-states:

- **Unconsidered**: Objects that were previously considered, but have been unconsidered.
- **Sensed**: Objects that are currently being sensed by the sensory system.
- **Unknown**: Objects that are not in memory and are not being sensed.

- **Considered**: A *considered* object is an object about which a decision has not yet been made. There are two testable sub-states:
  - **New-Considered**: An object was initially constructed in memory from a transform or an activator.
  - **Reconsidered**: An object was previously activated and was *reconsidered.*
- **Activated**: An *activated* object is an object to which a commitment has been made. The semantics of activation vary by object. Table 5 defines the semantics for each class of HLSR object.

Objects in the activated state may have special semantics associated with them. The specific semantics implied by the activated state vary with each object type, and are summarized in Table 5.

Object	Activated Semantics
Goal	Available for association with a transform – a transform can be considered for the goal.
Transform	Transform body executes
Belief (or general memory object)	Is believed
Production Set	Productions within the set can fire
Preference Hint	The decision suggested by the hint is made by the ISA

Table 5: Activation semantics for HLSR objects.

HLSR constructs that are not declarative memory symbols are not directly associated with CCRU states.

2.5.3 Behavior Primitives for Sensing

Sensory system interactions vary not only with the underlying ISA, but also with the environment in which the behavior model is situated. Each ISA has its own mechanisms for interacting with a sensory system, but they have the same general form. That is, some bridge component (sometimes referred to as a simulation interface when a behavior model interfaces with a simulation) transforms information from the environment to a form suitable for representation in the ISA's memory. Sensed information is volatile in that it may be removed from memory at any time when the environment changes or the sensory system shifts attention.

HLSR leaves the details about how the sensory system interacts with the behavior model to the compiler, architecture, and external sensory system components. HLSR requires that sensed elements be converted to objects (symbols) in the same form as any other memory element. HLSR also requires sensed information to be typed as are other objects. This leads to a uniform memory model where the same mechanisms and constructs used to manipulate non-sensory objects can also be used to manipulate sensory objects.

Figure 14 shows how sensory objects are manipulated in HLSR. Any sensed object is placed in the **sensed** sub-state of the *latent* CCRU state. HLSR knowledge can test for this state in memory patterns just like for any other CCRU state.

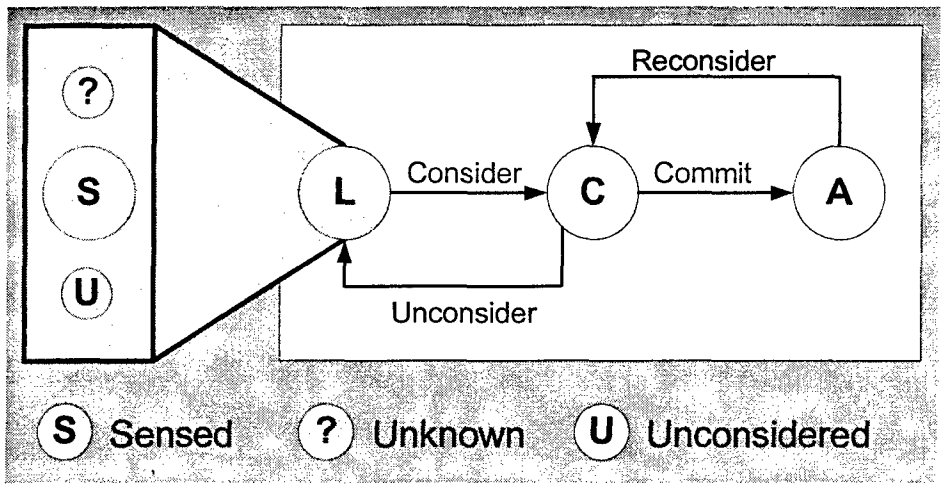


Figure 14: Augmenting the latent state with the "sensed" state to support the sensory system.

In part because of its volatile nature, most sensed information must be internalized. Internalizing sensed information involves copying part or all of the sensed information into memory so it is not lost when it is no longer sensed. This process is the source of a significant amount of low-level detail management in Soar and Act-R code (see Perceptual/Motor Interaction in Appendix A), and it is easy to for beginning knowledge engineers to get it wrong. HLSR provides an abstraction for the internalization process in the form of an **internalize** statement. The internalize statement is equivalent the consider CCRU process except that it considers a sensed object and all objects related to the sensed object. More details are provided in the *Specification of a High Level Symbolic Representation (HLSR) for Intelligent Systems*.

The sensory system ultimately controls the quantity and form of sensory system information; therefore, a behavior developer must interact with developers (if any) responsible for integrating the behavior model with an external system to understand what objects may be available for sensing in the behavior model.

2.5.4 Behavior Primitives for Motor Actions

Most behavior models are required to interact with some external environment. This implies that HLSR must have some mechanism for expressing and executing external actions. Soar and ACT-R interact with external environments using a motor system. The motor system is usually a combination of built in ISA processes and structures together with environment integration processes that are customized for each behavior model application.

Motor system interaction is a source of a large amount of low-level, error prone code in both ACT-R and Soar (see Perceptual/Motor Interaction and Output command structures & Proprioception in Appendix A). HLSR provides a higher level of abstraction that makes the behavior developer's responsibilities for managing motor interactions easier.

First, HLSR leaves most of the details of motor command execution up to the compiler and architecture. This includes the processes of output command status tracking and error reporting. From HLSR code, motor commands are executed through special object types called **command objects**. Command objects are treated in HLSR in the same way transforms are treated, with the exception that command object bodies (equivalent to transform bodies) are generated by the compiler or other external processes rather than the behavior developer. The relations defined within a command object form its parameters. The architecture and motor system track command execution state and update the command object with the same state information that transforms provide.

Commands are a form of *behavior primitive* in that they cannot be further decomposed into smaller sub-behaviors within HLSR. Behavior primitives are often application and domain specific. However, many applications share similar primitives, such as those to send messages

HLSR defines a set of standard primitives that are commonly used in behavior models. These primitives are described in Appendix B.

2.5.5 Failure Handling

A robust behavior model requires the ability to detect failures and respond to them. Both Soar and ACT-R have low-level mechanisms for detecting some classes of failures. Soar provides the *impasse* mechanism for detecting reasoning failures due to a lack of applicable knowledge. ACT-R has mechanisms for detecting retrieval errors from both memory and perceptual/motor modules, and also provides a special tag for productions that manage the failure. However, these processes are very low-level, and do not provide much high-level information about the semantics of and reasons for the failure.

To leverage these mechanisms effectively, behavior developers must design additional structures and processes on top of the built-in mechanisms. These processes are time consuming to build and maintain, and are often not built at all, leading to behavior models that are less robust than desired. HLSR helps make the process of failure management and resolution easier for a behavior developer by standardizing the failure management and resolution process and by specifying high-level knowledge and semantics with each failure.

Failures can be detected either by the architecture or by higher level processes defined within HLSR. In either case, HLSR constrains the failure detection and management process. HLSR also defines an explicit **failure object** construct for encapsulating knowledge about a failure. These constraints and failure objects allow HLSR knowledge to manipulate and recover from failures gracefully.

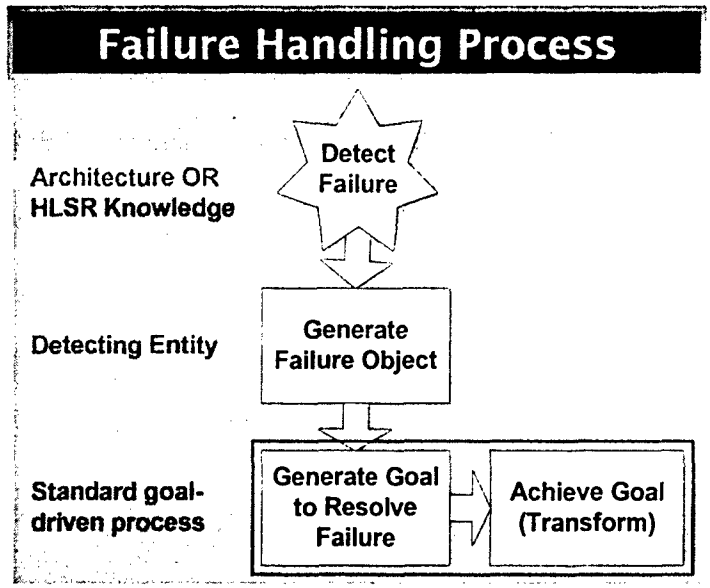


Figure 15: Error handling process in HLSR.

The HLSR failure handling process consists of a set of processing constraints shown in Figure 15. These constraints define processing steps that must be carried out by a combination of the ISA and HLSR knowledge:

1. Detect the failure.
2. Create and consider a failure object describing the failure.
3. Commit to the failure object.
4. Create and consider one or more goals to resolve the failure, referred to as failure resolution goals
5. Commit to one or more failure resolution goals
6. Create and consider one or more transforms, referred to as failure resolution transforms, to achieve the goal(s) committed in (2)
7. Commit to one or failure resolution transforms
8. Execute the failure resolution transforms until the goal(s) committed in (2) are met

If the failure is detected by the architecture or the sensory/motor systems, steps (1) and (2) are executed entirely by the architecture. The architecture makes all final decisions for commitment, and thus steps (3), (5), and (7) are executed by the architecture, but can be influenced by preferences. Steps (4), (6), and (8) define the failure resolution and recovery process. They are implemented in HLSR knowledge that reacts to the existence of the failure object and begins a failure resolution process that is encapsulated by the failure resolution transform(s).

Ultimately, the failure recovery process is defined by the behavior developer in one or more goals and transforms that reason about and resolves the failure. HLSR provides a standard interface and infrastructure for failure management and resolution. This interface and structure alleviate the need for various ad hoc failure detection processes and structures that often clutter behavior models.

HLSR provides a set of built-in failure objects that encapsulate common reasoning failures. These built-in failure objects as well as the details of the failure detection and management process can be found in *Specification of a High Level Symbolic Representation (HLSR) for Intelligent Systems*.

2.6 Compiler Design

HLSR requires a compiler to translate HLSR language statements into a form executable on the target architectures. During this effort, we defined requirements for the HLSR compiler, mapped HLSR constructs to specific representations in ACT-R and Soar, and explored in-depth the design of an HLSR-compliant compiler for Soar, *HLSR2Soar*. This section summarizes the compiler requirements and high-level design. *Requirements for an HLSR Compiler* presents the requirements and design in greater depth.

2.6.1 Compiler Requirements

This section summarizes basic requirements for any HLSR compiler. The high level requirements are:

- Given an HLSR source code program, verify syntax, and generate native code for execution on a ISA (Soar or ACT-R).
- The compiled program should be able to run on the chosen ISA; however, there will be a number of speed, space and fidelity tradeoffs that any compiler makes when implementing a target program from HLSR.
- The compiler should be written in a modern, high level language such as Java or C++.
- A compiler-compiler tool (e.g., yacc (yet another compiler-compiler) and descendants) should be used to generate the general parser code from a grammar specification

Parser requirements:

- The grammar specification should be in Backus-Naur Form (BNF) or extended BNF (EBNF). This requirement ensures that standard tools can be used to parse HLSR.
- Syntax errors in behavior specification should be reported to the user
- The output of the parser should be represented in an intermediate form. Initially, we assume the intermediate form is a parse tree, which satisfies this requirement.

Code generator requirements:

- The code generator should accept as input an syntactically verified HLSR program, represented in the intermediate representation
- A mapping of HLSR processes and representations to specific constructs in the target ISA representation. The mapping may require a model of the target architecture.
- An ISA runtime library, which manipulates and tags program data structures according to the constraints of HLSR.
- The output of the code generator will be a program executable on a specific ISA.

2.6.2 Compiler High-Level Design

The proposed HLSR compiler architecture is illustrated in Figure 16. The two main components are the parser and the code generator. Details of each of these modules are described in the following sub-sections. This effort focused primarily on the Code Generation module, and specifically mappings from HLSR source to the target architectures, ACT-R and Soar.

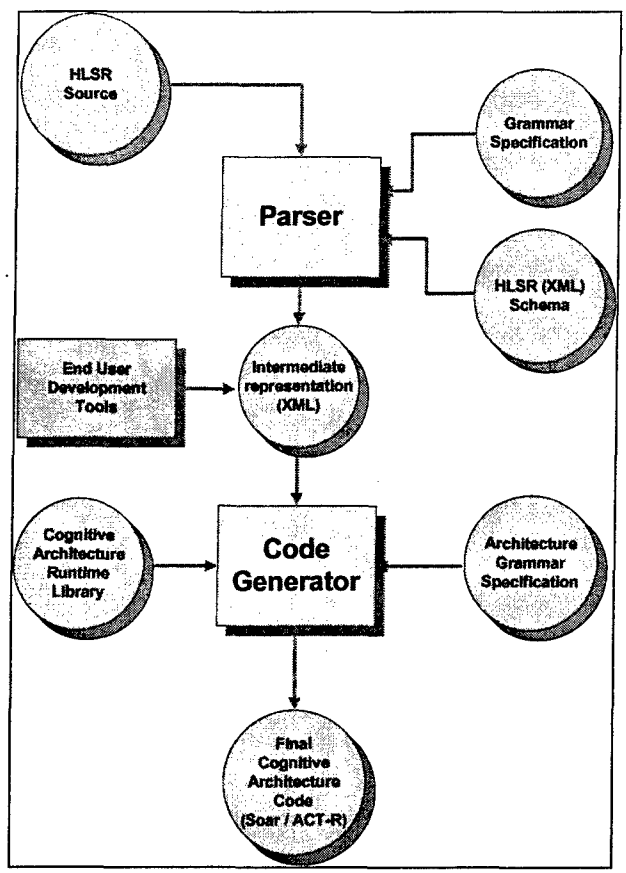


Figure 16: The HLSR compiler architecture.

2.6.2.1 Parser Design

The first major component is the parser. Parsing is the process of determining if a string of tokens is a legal sentence in some specified grammar. The HLSR parser will construct a tree representation of the input HLSR code. When a parse tree cannot be constructed, this signals and error in the input code.

A parser can be constructed for any regular grammar and, importantly, is ISA independent. Thus, the first stage of processing in an HLSR compiler should not be specific to any target architecture. The parsing process consists of four main tasks, lexical analysis, syntax analysis, semantic analysis and intermediate code generation. These components are detailed in *Requirements for an HLSR Compiler.*

2.6.2.2 Code Generator Design

The code generation component consists of converting the intermediate code generated by the parser into a form executable for the specified ISA. A critical requirement for code generation is a mapping between the high level representations and processes of HLSR, and the specific mechanisms and representations of an ISA. This effort focused on two specific architectures, ACT-R and Soar, and defined mappings directly from HLSR to each of these architectures. This section briefly summarizes the specific mappings developed in Year 1 for Soar and ACT-R. *Requirements for an HLSR Compiler* discusses these mappings in more detail, including tradeoffs in the specific choices, and a detailed mapping and design for an HLSR2Soar compiler.

Table 6 lists some of the mappings developed during the course of the project. For example, beliefs are represented in ACT-R as declarative chunks and are activated by way of their assertion in a buffer and retrieval using ACT-R's spreading activation mechanism. In

contrast, Soar beliefs are represented as objects in Soar's working memory (beliefs are actually organized into named collections in working memory; see *Requirements for an HLSR Compiler* for details) and Soar's decision cycle is used to activate beliefs.

HLSR construct	Mapping to ACT-R	Mapping to Soar
Activator / Terminator	R: ACT-R Production Rule A: Conflict Resolution Cycle	R: Soar Production A: JTMS
Belief	R: Declarative Memory Chunk A: Buffer Creation + Spread. Act	R: Declarative Memory A: Decision Cycle (DC)
Goal	R: Goal Buffer + Declarative Memory A: Production Rule + Retrieval	R: Declarative Memory A: Decision Cycle
Manipulator	R: Production Rules + Commands A: Conflict Resolution + Buffer	R: Soar operator A: Decision cycle
Preference	R: Production Utilities A: Conflict Resolution Cycle	R: Soar Productions A: Decision Cycle + Learning
Production Set	R: Type-Specific Production Set A: Conflict Resolution	R: Declarative Memory **(problem space)** A: Decision Cycle
Query	R: Sequence of Retrievals/Tests A: Conflict Resolution Cycles	R: Soar elaboration A: JTMS
Transform	R: Set of Production Rules A: Conflict Resolution Cycles	R: Declarative Memory A: Multiple Decision Cycles

Table 6: Mappings from HLSR constructs to ACT-R and Soar. "R" indicates the data structure used to Represent the construct, "A" indicates the mechanism in the architecture by which HLSR Activation is achieved.

2.6.2.3 Run-time Libraries

Figure 16 includes a run-time library as a component of code generation. . These libraries enable the mappings from HLSR to the ISA in those situations where an HLSR construct is not directly supported by the architecture. For example, we chose to map HLSR goals in Soar to structures in declarative memory, rather than Soar's low-level impasse goals (*Requirements for an HLSR Compiler* discusses this decision in more detail.). However, that decision requires some additional knowledge (represented as Soar productions) to monitor for goal achievement, goal failure, etc. because the mapping does not leverage the inherent capabilities of the architecture for this functionality.

Figure 17 presents a schematic production that will be present in the HLSR run-time library for Soar. It simply says that if a (HLSR) goal in Soar memory is an activated achievement goal and the conditions specified by the developer that indicate when the goal is achieved (InternalIsMet) have actually been achieved, that the goal's status is marked as "achieved". Other components of the HLSR run-time for Soar would then act to reconsider the goal (because HLSR requires the immediate reconsideration of any achievement goal when it has been achieved). The run-time library thus acts as an enabler for the execution of HLSR in each ISA. The run-time library is not generated by the compiler but is hand-coded. However, the run-time library is a component of the compiler, and needs to be written only once for each target architecture.

```
HLSR-RTL: GOAL-ACHIEVED
IF
  a goal is an achievement goal
  the goal is activated
  the goal's internal-is-met conditions are satisfied
THEN
  the goal's status is achieved
```

Figure 17: An example production in the HLSR run-time library for Soar. This production marks the status of an HLSR achievement goal as "achieved" when the goal's "Internal Is Met" conditions have been satisfied.

2.6.3 Reference Models

Agent knowledge implemented for a particular ISA should be reusable from one application to another, reducing development costs over multiple projects. In practice, however,

reuse of knowledge and transfer of validation is limited. Limited reuse results from the large gap between the high-level specification of the architecture and its actual implementation and the complexity of the ISA software systems, which evolve with advances in research and new implementation technologies. The lack of specification and difficulties with reuse are particularly acute problems for HLSR because both the compiler and the run-time library are dependent on the specific version of the ISA.

We explored the feasibility and technical hurdles for creating formal models of ISAs such as Soar and ACT-R and built a preliminary Soar model. The model was defined within an ontology representation tool that provides a foundation for the definition of semantically well-defined classes and relations with which to describe architecture components (representations and processes), their interrelationships, and their connection to software engineering and artificial intelligence concepts [43]. More details of the model are discussed in *Towards Formal Models of Cognitive Architectures.* This section summarizes the model we developed.

We defined Soar components and relationships, creating an initial high-level ontology with instances corresponding to the processes and representations of Soar. The current model consists of 66 classes and 51 property slots. We followed a common ontology representation methodology [32]. Because ontologies can quickly become "theories of everything," we focused our design on some key competency questions related to the articulation of one of the core ideas of HLSR, the CCRU processes, and the initial design of HLSR2Soar. Thus, the model is by no means complete, but does provide answers to some many HLSR-relevant *competency questions* [32].

Figure 18 provides a view of the ontology showing some of the concepts in the model (Algorithm, Intention) and components of Soar that map to these concepts (*instantiation support, selected operator).*

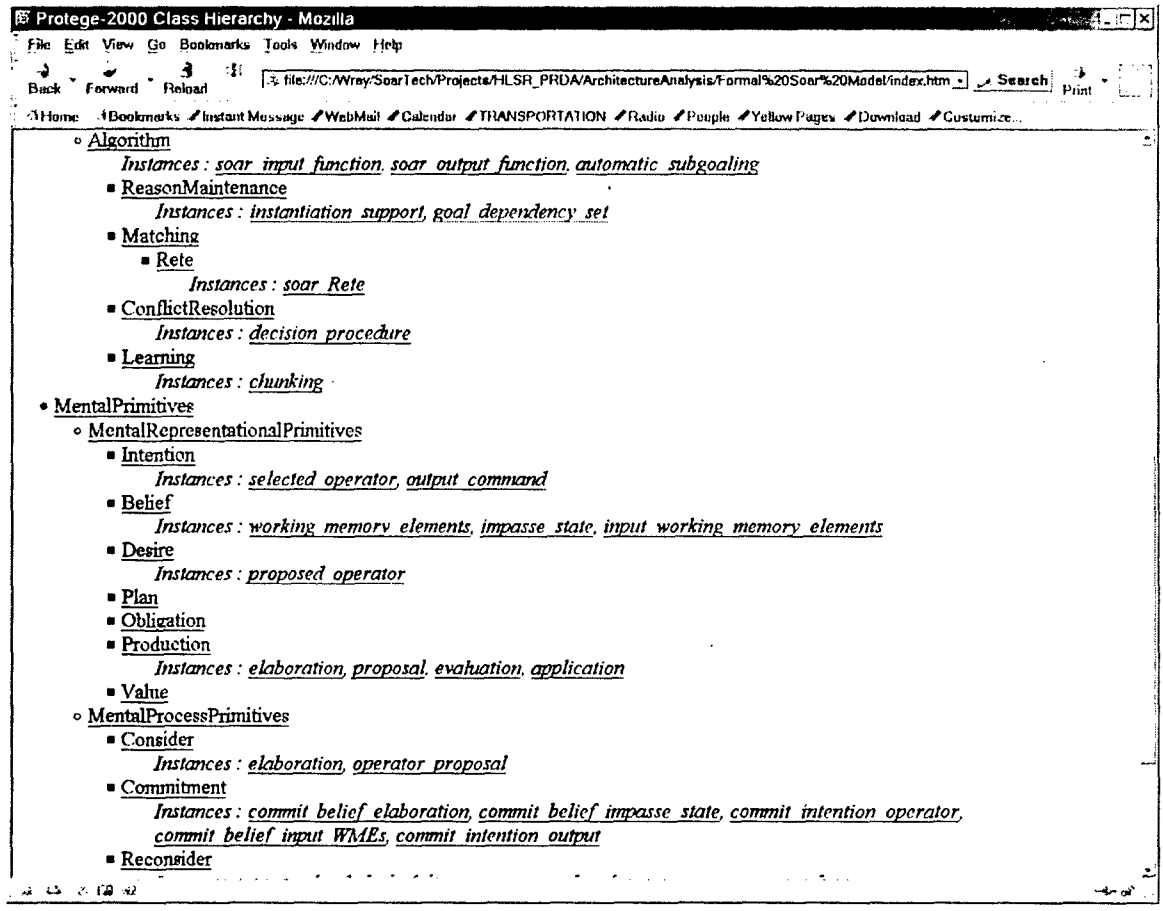


Figure 18: This example shows a partial view of the class hierarchy. We represented HLSR representational and process primitives such as intention and commitment and mapped Soar objects to these concepts.

3 Feasibility Demonstration

An important goal of this effort was to prove the feasibility of the HLSR concept. Given the limited time frame of this first year effort, it was not possible to build full scale compilers and thus prove feasibility by construction. Instead, our approach consisted of the following two steps:

1. Design mappings from HLSR constructs and constraints into Soar and ACT-R constructs and processes. This process and the results were discussed in Section 2.5.
2. Implement a small scale example that shows HLSR code compiled to Soar and ACT-R code. This process is the focus of this section.

The first step shows that, in general, there is a plausible mapping from HLSR to ACT-R and Soar. The second step shows that the implementation details of HLSR are compatible with each ISA and can be implemented reasonably efficiently (though efficiency was not a primary concern for these examples). This approach does not prove feasibility altogether, both because of the limited scale of the example and because we compiled HLSR "by hand" (that is, manually translating the HLSR code rather than using an automated compiler) to demonstrate functionality. Within these constraints our demonstration was successful and indicated that no fundamental road blocks to implementation are present and that the translation process can be accomplished for the core HLSR constructs (i.e., those that are exercised in the example).

The rest of this section describes the implementation of a simple example from design to Soar and ACT-R code.

3.1 Tools and Editing Environments

Any programming language, of which HLSR is specialized version, requires tools and editing environments to allow engineers to leverage their full capabilities. It was not in the scope of this effort to build HLSR tools; however, it was part of our plan to investigate and evaluate viable options for HLSR support tools.

We anticipate two classes of user for HLSR. The first class of user is the knowledge engineer who will build and maintain behavior models directly using HLSR. The second class of user is the end-user tool developer who will use HLSR as a "back-end" representation for storing knowledge generated by the tool. Each of these users requires different interfaces and support.

The knowledge engineer typically interacts with encoded knowledge using a development environment. The best environments integrate editing, browsing, execution, and debugging into one system. These are often referred to as integrated development environments (IDEs).

We developed a mockup version of such an IDE that included only editing capabilities. We used the Visual Slickedit IDE (http://www.slickedit.com/) as a base system, and customized its lexing and syntax highlighting capabilities to handle HLSR knowledge. We used this customized Visual Slickedit configuration, shown in Figure 19, to create the HLSR examples for the feasibility demonstration.

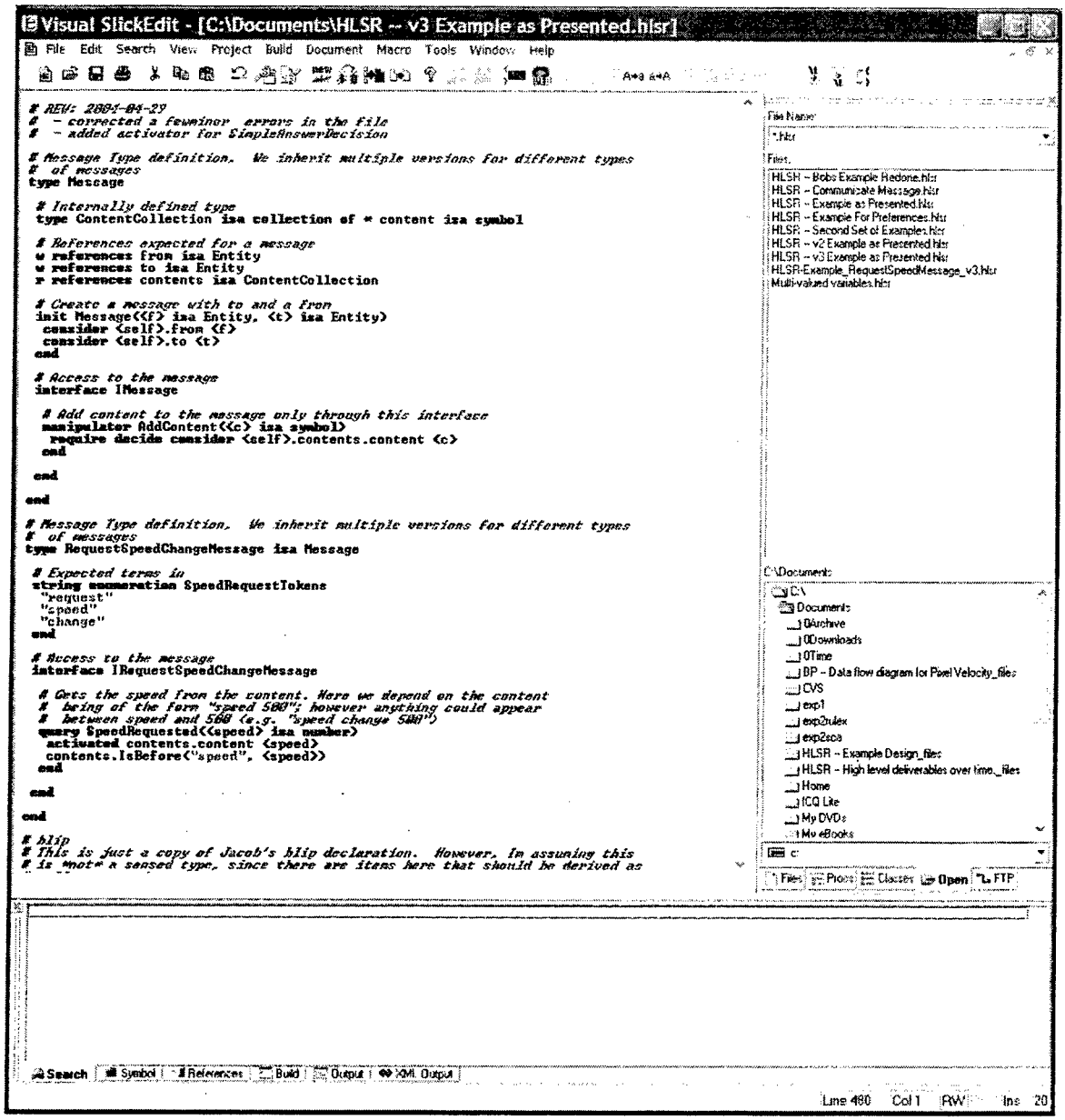


Figure 19: Visual Slickedit environment customized for HLSR. This is an example of an IDE such as will be required for knowledge engineers using HLSR.

IDEs are difficult and labor intensive to build; however, recent work in the open source community has led to the development of IDE frameworks that supply the fundamental IDE capabilities. These frameworks strongly support extension through "plugins" and other

mechanism, and are being used in traditional SE as the core tool around which many IDEs are built.

The Eclipse project (http://www.eclipse.org/) provides one such framework and it has been successfully leveraged to build a prototype editor for Soar. We propose to base HLSR tools on the Eclipse framework as is being done for Soar. This will greatly reduce the effort required to build the IDE, and encourage faster adoption of HLSR due to development tool support.

The other class of HLSR user, the end-user tool developer, typically requires a program-friendly (i.e. computer-friendly) interface into a representation. End-user tool developers do not usually write and manage back end representations, but rather, write systems that input, output, and transform such representations.

The current standard for program data storage is the Extensible Markup Language (XML). We intend to implement an intermediate form of HLSR in an XML format. This will enable end-user tool developers to use HLSR with standard XML parsers, and will allow format and syntax checking supplied by XML schemas. The XML intermediate form, and its further advantages are discussed in *Requirements for an HLSR Compiler*.

3.2 The AMBR Example

For the feasibility demonstration, we selected a behavior representation task from the Agent-based Modeling and Behavior Representation (AMBR) program [18, 35]. One of the goals of the AMBR program was to compare the capabilities and characteristics of models built with different ISAs. An overview of the specific AMBR task is provided in section 3.2.1. The AMBR example was selected because it has several desirable characteristics:

1. Our team is familiar with the problem and had implemented both Soar and ACT-R models for AMBR previously.

2. The required functional behavior is straightforward and relatively simple to understand and implement.
3. The task, while simple, requires dynamic decision making based on conflicting priorities and knowledge, and thus addresses some of the unique requirements of behavior modeling.

HLSR is a tool for bridging the gap between design and implementation code. Therefore, it is important that we create an example that exercises the development steps with which HLSR interacts. These steps are the design phase, the implementation phase, the testing/debugging phase, and the maintenance phase. The latter two phases are difficult to test without building a large system over time using a set of well-developed tools; therefore we were unable to test them in this effort. However, our approach incorporated both design and implementation, thus allowing us to do a preliminary analysis on how HLSR affects the development process.

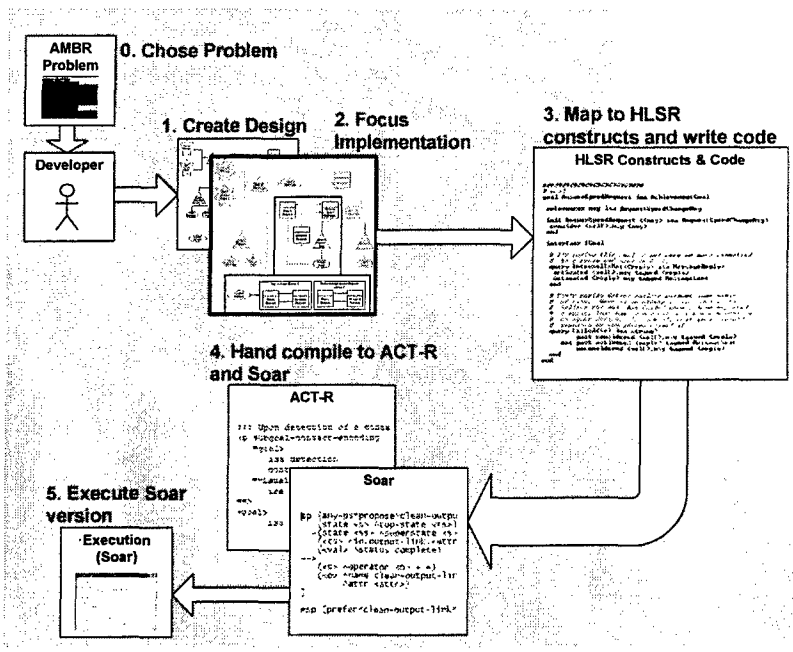


Figure 20: Development approach used for AMBR example.

We used a five step development process that is diagramed in Figure 20. The preliminary step involved choosing the example, which is discussed above. Our first development step was creating a design. HLSR should map well to design constructs thus reducing the amount of effort a developer must use to map design constructs to code. Our second step involved selecting a portion of the design for a detailed implementation. A complete implementation was beyond the scope of this effort, so we selected a portion of the task for detailed implementation, as described in section 3.2.1. Our third step was to design HLSR code to implement the example. Several attempts and edits were made on this code over time to reflect changes to the HLSR specification, and improved implementation technique. Our fourth step was to hand compile the HLSR code to ACT-R and Soar code. This hand compiling process followed the mappings described in 2.6.2.2, with additional details not covered in the compiler mapping documents dealt with on a case by case basis using best practices for the given architecture. Finally, the Soar code is complete enough for execution, and thus we executed it to make preliminary performance measurements. The ACT-R code is executable, but the extra data structure and detail setup required to execute it was not implemented due to project time constraints.

The final code for this example is available as Appendix C, D, and E of this report.

3.2.1 AMBR Problem Specification

The AMBR task simulates one (simplified) task that an air-traffic controller (ATC) might perform. The ATC is assigned an airspace and monitors the aircraft in the assigned airspace. The monitor is represented on a simulated radar screen. There are two primary task conditions in AMBR, *aided* and *unaided.* In the aided task, the representations of planes on the screen are color-coded to indicate a task that needs to be performed on them. In the unaided task the representations of planes are not color-coded, and the ATC must evaluate the screen and

individual aircraft to determine which task needs to be performed. For the feasibility demonstration, we used the unaided task. That is, the behavior model implementations ignored colors. The ATC tasks for both the aided and unaided problems include:

- Accepting an aircraft into the ATC-controlled airspace (when contacted by another ATC),
- Welcoming a aircraft
- Transferring an aircraft to another ATC
- Contacting another ATC (to complete the transfer)
- Deciding whether to accept or reject a speed change request

The challenge in the task is that there are many planes, and any of them may require attention at any one time. Penalty points are assigned both to incorrect actions and to delays, so the user has a sense of urgency in performing the task. All tasks are performed via the manipulation of a GUI, shown in Figure 21, requiring interaction with an outside world.

A message buffer on the right-hand side of the display provides a record of activity. A user can inspect this buffer (called a message history list, MHL) to determine what operations have previously been performed on a blip (because there are penalties for taking incorrect action, inspecting the buffer is important to ensure that the right action is taken).

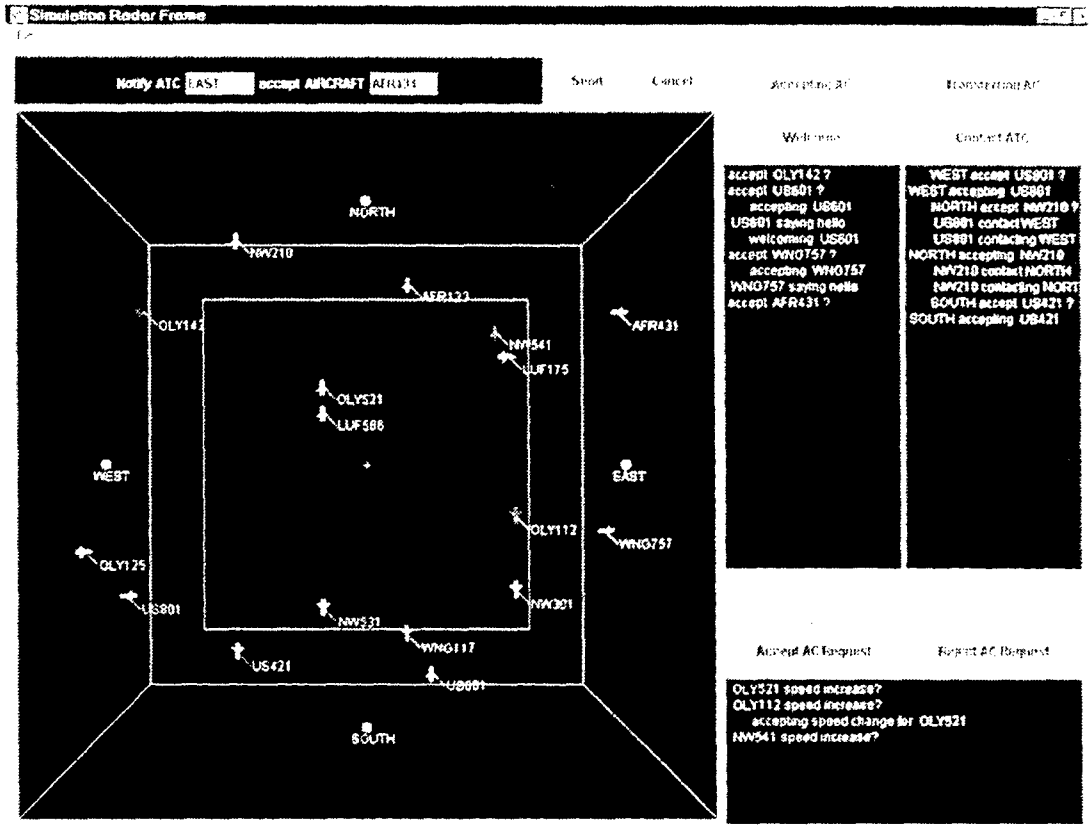


Figure 21: AMBR air traffic control screen.

For the feasibility demonstration, we focused on only one small segment of the AMBR task – responding to a request for a speed change by a contact. The rules for this task are simple:

- Reject a speed change request if another plane is in front of the requesting plane.
- Otherwise, accept the speed change request.

Several aspects of this task made it desirable as an HLSR example.

1. It was simple enough to complete within the scope of this effort
2. It involved reacting to an external stimulus (the speed request response).
3. It involved interaction with the motor system (sending the response)
4. It involved making a decision based on context.

5. When considered in the context of all of the other decisions that must be made, it required goal driven behavior to achieve behavior consistency and ensure that the task is executed to completion.

The design and implementation of this task are covered in the following sections.

3.2.2 AMBR Problem Design

There is no standard design methodology or process for behavior modeling. We selected the agent oriented programming methodology (AOP) Prometheus [33] as the basis for our design. We augmented this methodology with additional features necessary for our behavior model. We selected Prometheus for the following reasons:

1. It has sufficient documentation for picking up and using quickly.
2. Though not a behavior modeling design methodology, it shared many characteristics with behavior modeling processes, and thus mapped well to our problem.
3. It provided diagrammatic structures and formalisms for representing design, thus alleviating the need for us to invent them.
4. It is currently being used to build agent-based systems; that is, it is not "dead" methodology.

Figure 22 is an interaction diagram that shows important behavior model elements and how they interact (Figure 23 contains the key for Figure 22). It follows the path of receiving a message and processing two message variations: a request for speed change, and a request for aircraft transfer. We focus our attention on the portions of the behavior inside of the boxes. This includes the request for speed change behavior and preferences for this behavior.

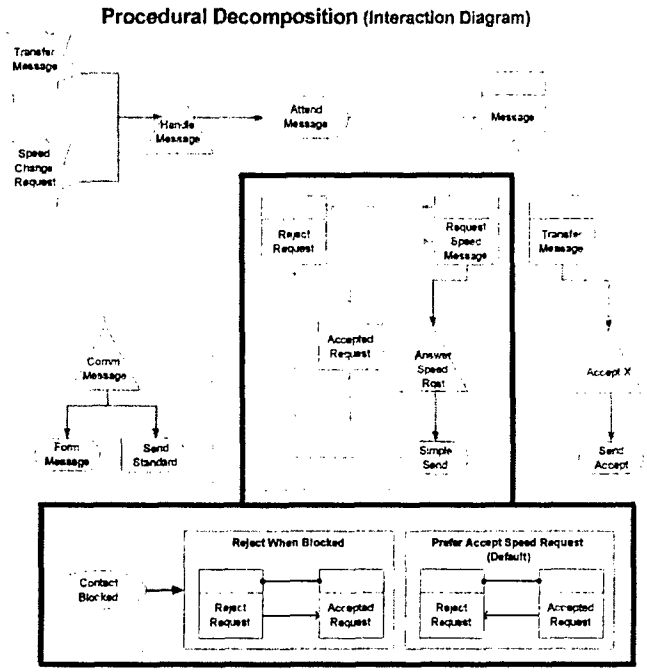


Figure 22: Design for the Answer Speed Request and related tasks using a modified Prometheus methodology.

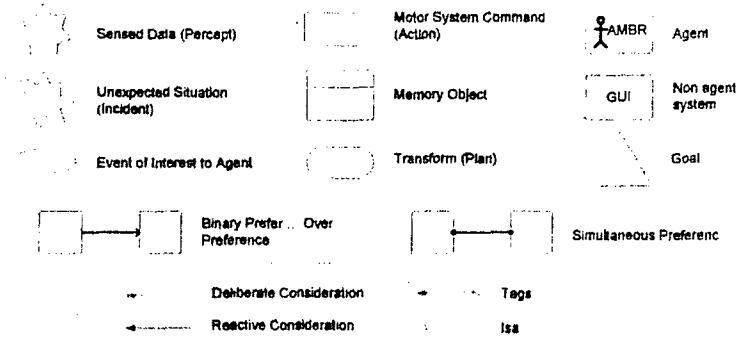


Figure 23: Key for Figure 22.

The behavior for processing a speed request message is triggered by the existence in memory of a *Request Speed Message* that has not yet been processed. The rectangular boxes together with the arrows show a portion of the taxonomy of messages in the system. The red arrow pointing into the *Answer Speed Request* goal indicates that this goal is reactively considered when a *Request Speed Message* is committed. The red arrow pointing into the *Simple*

Send transform indicates that this transform is reactively considered to achieve the *Answer Speed Request* goal. The *Simple Send* transform proposes a choice to either accept or reject the request. This choice is formed as the consideration of two tags on the *Request Speed Message*: *Reject Request* or *Accept Request*. The preferences diagramed in the lower box, influence the choice of which tag is committed. If the aircraft is blocked in front, then it is preferred to reject the request. If the aircraft is not blocked, then it is preferred to accept the response. Finally, the *Simple Send* transform considers the *Send Message* goal to reply to the requesting aircraft with either the accept or reject response.

The diagram in Figure 22 emphasizes interaction between components, but should not be interpreted as implying that behavior is sequential. Static diagrams are not well suited for expressing the dynamic features such as conflict resolution, thus these aspects of behavior are often not adequately described in designs. Other design decompositions and perspectives can be used to emphasize other dependencies and interactions within a behavior. Some of these are provided in the *HLSR Feasibility Demonstration Design and Code Package*.

3.2.3 HLSR Code

One important goal of HLSR was to make it easy to map design constructs to HLSR. Looking at the diagram in Figure 24, we see that this goal has been met. The yellow boxes indicate constructs in the diagram that map directly to HLSR constructs. While we added the preferences and the tagging operation to the Prometheus methodology, the remaining elements of the design are based directly on the Prometheus methodology. Therefore, it is clear that our design is not custom built to map to HLSR.

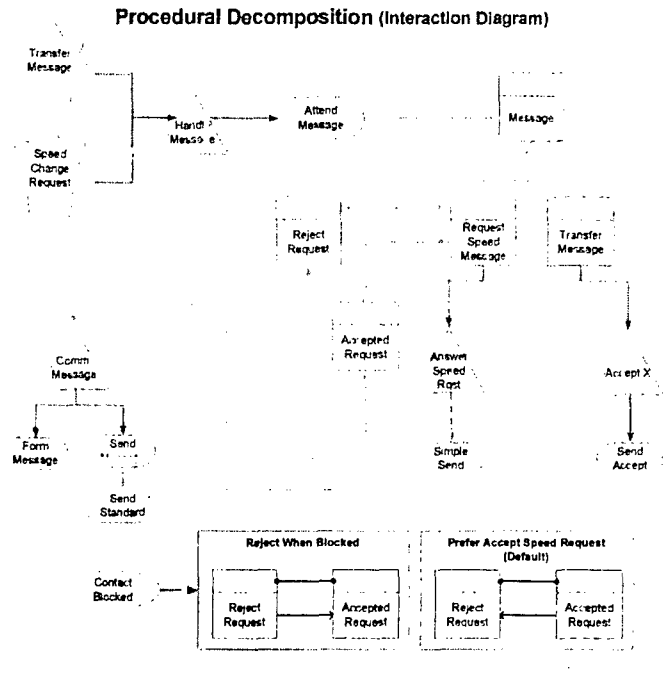


Figure 24: Mapping HLSR constructs onto the design.

The complete HLSR code for this example is supplied in Appendix C. However, here we provide an overview of the mapping of design to HLSR code. Note that as each design construct is mapped to HLSR, its primary structure remains the same, but many details are added. The design becomes the framework within which implementation details are added. This is an important organizational structure for making large scale development efficient and maintainable because it helps the developer encapsulate and track details at appropriate levels of abstraction. The design captures the highest level structure and architecture. The encoded HLSR knowledge captures the details of design.

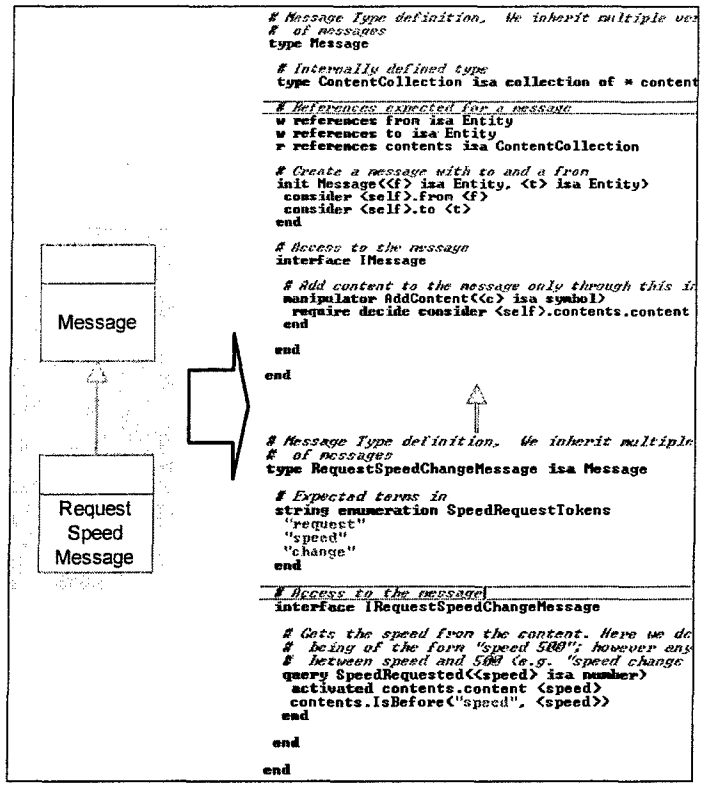


Figure 25: Mapping taxonomy to type definitions.

Figure 25 shows a portion of the message taxonomy included in the design diagram. Object taxonomies map directly to HLSR type definitions. Here we see a definition of the *Message* type and a derived *RequestSpeedChangeMessage* type. It is important to remember that the behavior model can reason over this taxonomy information as necessary to determine what type of message it is processing and what each of its relations are (see 2.5.1.3). This structure also serves as documentation of the behavior model's declarative knowledge, and can be browsed by developers that are maintaining the model.

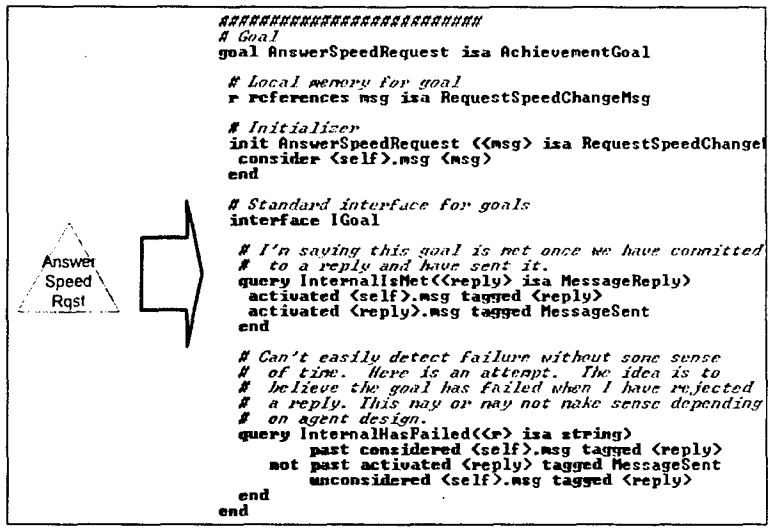


Figure 26: Mapping goals to HLSR.

Figure 26 shows that goals map directly to HLSR goal constructs. We see in the HLSR code that the *InternalIsMet* and *InternalHasFailed* queries are filled out indicating when the goal is met and failed respectively. This important information is encapsulated with the goal definition unlike in Soar and ACT-R where it is tangled with productions that manipulate the goal.

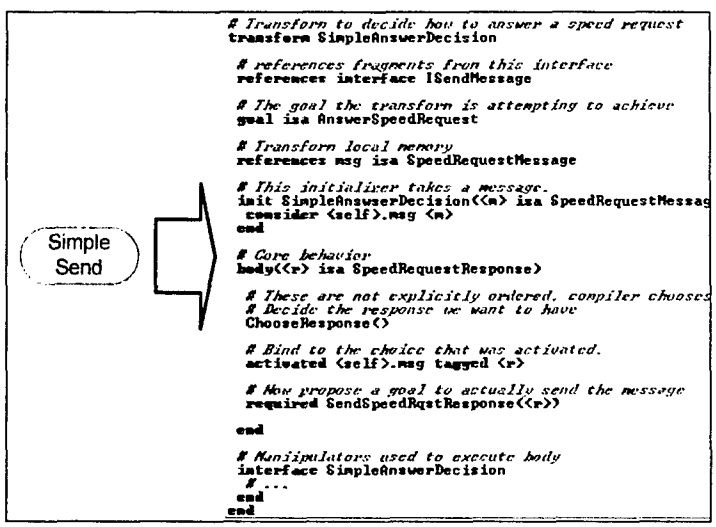
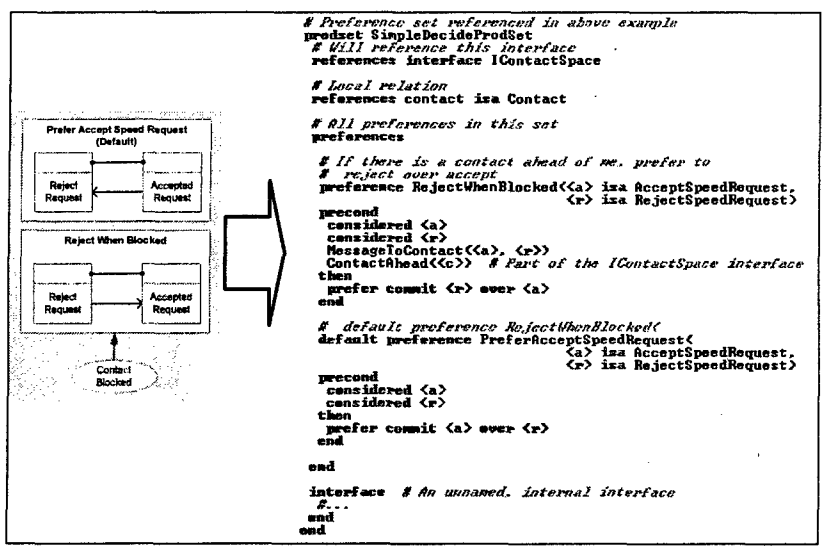


Figure 27: Mapping transforms to HLSR.

Figure 27 shows the transforms map directly to HLSR transforms. Important details of the transform include the transform goal statement ("goal isa AnswerSpeedRequest") and the body which executes after the transform is committed. Here we see that the transform is intended only to achieve AnswerSpeedRequest goals, but it is possible to create transforms that attempt to achieve any class of goal.

The transform body expresses process constraints. The ChooseResponse manipulator encapsulates the process of tagging the speed request message with two possible responses. The memory pattern (i.e. "activated <self>...") tests the message for an activated tag that indicates the response the model has selected. Finally, the last statement in the transform body invokes a manipulator to actually send the response. This statement is *required*; that is, it must complete successfully if the transform is to complete successfully. The required statements are used by the compiler and architecture to track transform success and failure, which are useful for automatic tagging and process tracking (section 2.3.3.3). Other transform details are shown in Appendix C.




```
# Preference set referenced in above example
predset SimpleDecideProdSet
 # Will reference this interface
 references interface IContactSpace

 # Local relation
 references contact isa Contact

 # All preferences in this set
 preferences

  # If there is a contact ahead of me, prefer to
  # reject over accept
  preference RejectWhenBlocked(<a> isa AcceptSpeedRequest,
                               <r> isa RejectSpeedRequest)
  precond
   considered <a>
   considered <r>
   MessageToContact(<a>, <r>)
   ContactAhead(<c>)  # Part of the IContactSpace interface
  then
   prefer commit <r> over <a>
  end

  # default preference RejectWhenBlocked(
  default preference PreferAcceptSpeedRequest(
                               <a> isa AcceptSpeedRequest,
                               <r> isa RejectSpeedRequest)
  precond
   considered <a>
   considered <r>
  then
   prefer commit <a> over <r>
  end

 end

 interface  # An unnamed, internal interface
  #...
 end
end
```

Figure 28: Mapping preferences to HLSR.

Figure 28 shows that the preferences in the design map to HLSR preference constructs encapsulated within a production set. Here the production set encapsulates two preferences. The first preference hints that a *RejectSpeedRequest* tag should be preferred over an *AcceptSpeedRequest* tag when another contact is ahead of the requesting contact. The second preference indicates that the default choice should be to commit the *AcceptSpeedRequest* tag.

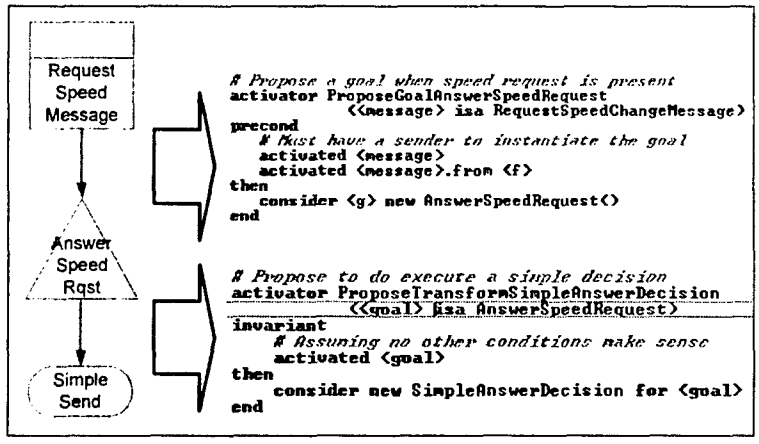


Figure 29: Mapping reactive consideration arrows to HLSR.

Figure 29 shows that the design diagram's red dependency arrows map directly to activators in HLSR. Two activators are shown here. The first considers a new *AnswerSpeedRequest* goal whenever a *RequestSpeedMessage* is activated. The second considers the transform *SimpleAnswerDecision* (a rename of *SimpleSend*) to achieve the *AnswerSpeedRequest* goal.

Not only do the examples show how HLSR maps well to design constructs, but also that HLSR is significantly more readable than native Soar and ACT-R code (as shown in Appendices D and E). Readability helps support software engineers with shallow knowledge of specific ISAs to build and maintain behavior models.

3.2.4 Alternative approaches to mapping

In Section 2.6, we described mappings from HLSR to both ACT-R and Soar. The approaches we took to thee mappings represent two alternative approaches to executing HLSR programs in the cognitive architectures. For the Soar mappings, we chose to develop complete mappings that would guarantee a functional Soar program that closely adhered to the HLSR behavior specification. However, this mapping introduces "gaps" in which Soar's mechanisms and process do not (appear to) provide the complete functionality required by HLSR. For the mappings to ACT-R, we chose to focus on a direct mapping from HLSR to architectural process and representation. In some cases, these mappings may not be able to guarantee the same kind of functionality (or even complete functionality) as required by HLSR. However, the mappings are strongly motivated by the specific mechanisms of the architecture.

In effect, the alternative approaches to the mappings reflect a top down (HLSR-to-architecture) bias in the Soar mappings and a bottom up (architecture-to-HLSR) bias in ACT-R's mappings. Both approaches are important in understanding the requirements and limitations on compilation. For example, the top down mappings satisfy basic functional and performance requirements, but at the expense of not fully leveraging the architecture. The "gap" between the HLSR specification of HLSR and the Soar mappings require the articulation of a computational approach to bridge it. In the next section we introduce "micro-theories" to supplement the HLSR-to-Soar mappings. The HLSR2Soar run-time library is effectively a realization of these micro-theories.

The bottom up approach, as exemplified by the ACT-R mappings, strongly leverages the architecture. However, because the mappings so rigorously adhere to the architecture, it may be impossible to guarantee that all legal HLSR programs may be executed successfully by the

compiler/architecture. More realistically, when bottom up mappings are used, it will be likely that an HLSR behavior developer will need to understand the underlying architecture in more detail than when top down mappings are used, in order to ensure that the HLSR program can execute on the architecture.

This competition, between leveraging the architecture and abstracting from it, represents one of the fundamental tensions in the effort to create a higher level language for behavior specification. The top-down mapping approach essentially views HLSR as a distinct architecture. The run-time library provides an on-line translation of the higher level architecture constructs to the lower level architecture, comparable to the relationship between interpreted Basic program code and its execution on a specific hardware architecture. The bottom-up approach attempts to define layers of abstraction above the architectures. The primary challenge of this approach is to abstract just enough that the architectures can be leveraged to the greatest extent possible.

Exploring the specific trade offs and consequences of these different approaches to mapping is a subject of future work. However, we optimistically note a third approach that is not available in more traditional computational systems. Because both architectures support knowledge compilation [3, 27, 37, 41], it may be possible, in the long term, to define HLSR mappings in such a way that the intermediate bridging steps (i.e., the run-time libraries) could be gradually eliminated from the run-time consideration of the agent. This learning would enable knowledge specification in the "high level" HLSR language, but (eventual) execution in unmediated forms on the target platforms, fully and naturally leveraging the specific architecture.

The theories of both Soar and ACT-R support this progression via chunking [27] and production composition [41]. In practice, both mechanisms have weaknesses that limit their

robust use in general systems. HLSR may encourage additional research and development of these mechanisms. Further, because the patterns of execution generated by the compiler will be more constrained and predictable than the variation in hand-generated code, it may also be possible to structure the HLSR compiler to avoid known problems in the learning mechanisms (e.g., some recent research has been directly motivated by problems in chunking [47, 49]). The HLSR micro-theories for Soar reflect this goal.

The following sections introduce the micro-theories we developed for the HLSR2Soar compiler and then detail the implementation of these micro-theories in the HLSR run-time library for Soar. While tedious and labor-intensive, the exploration of a compiler for the top down (Soar) mappings was relatively straightforward. However, determining appropriate approaches to the compilation of bottom up (ACT-R) mappings will require significantly more research. Without micro-theories (and the run-time library which realizes them), there will be many more constraints on the compilation process which have to be considered. During this initial phase of exploration, we chose to more fully evaluate the (more or less straightforward) compilation from top down mappings. Appendix E outlines what kind of ACT-R program a bottom up compilation approach might provide.

3.2.5 Results of Feasibility Evaluation

In order to evaluate the feasibility of HLSR, we wanted to show that HLSR programs could be compiled to Soar and ACT-R but we needed to do so without building a full compiler. As discussed above, we elaborated the HLSR-to-architecture mappings (see Section 2.6.2) and defined "micro-theories" that detailed how mappings would actually be realized in Soar systems. The micro-theories specify the mappings at a very detailed level.

```
top-state.goals (identifier)
  goal (identifier)
    name (string)
    goal-type enum(achievement, maintenance)
    goal-subtype (string)
    status enum (not-achieved-or-failed, achieved, failed)
    supergoal (pointer to a single(??) prior goal)
    subgoal (pointers to subgoals used in pursuit of goal)
    tags (identifier: general collection of goal tags)
      transform (identifier)
      (all sub-structure associated with particular transforms)
      internal-is-met enum(satisfied, not-satisfied)
      internal-is-failed enum(satisfied, not-satisfied)
      HLSR-state enum(new-considered, activated, reconsidered)
```

```
TEMPLATE: CONSIDER-NEW-GOAL
IF
    Activation conditions satisfied (from activator)
THEN
   create new goal of specified type (from HLSR code)
   annotate goal with any initialization parameters
            tag: HLSR-state new-considered
```

Figure 30: Examples from the Soar micro-theory of HLSR goal representation. The top section indicates how HLSR goals are represented in Soar declarative memory. The bottom section illustrates a compiler template that defines how information from a goal's activator should be represented in Soar. The micro-theory theory also specifies run-time library functionality, such as the production shown in Figure 17.

For example Figure 30 illustrates some of the micro-theory defining HLSR goal processing in Soar. The micro-theory specifies how an HLSR construct will be represented in Soar (the top section) and how individual Soar productions can be generated from HLSR statements (the compiler template in the bottom section). The micro-theory also specifies any needed functionality in the run-time library. For example, for HLSR goals in Soar, the micro-theory defines a number of productions that monitor for goal achievement and failure. One of the run-time productions from the HLSR goals Soar micro-theory is illustrated in Figure 17. The specific details of the micro-theories developed for Soar for the feasibility demonstration can be found in *Requirements for an HLSR Compiler.*

Having defined the micro-theories for each Soar construct, we used the micro-theories to translate the HLSR code for the AMBR speed request response example (as described in Section 3.2.3 and presented, in full, in Appendix C) to Soar. Initially, we executed this process manually, but towards the end of the project we also automated the generation of Soar productions for the AMBR example (see the next section for those details).

Table 7 summarizes the results of the feasibility testing. Each row represents the total production knowledge (second column) and a number of performance metrics for a Soar agent. The "AMBR Soar" row shows the results from a hand-coded AMBR agent (i.e., one that is written directly in Soar). The second row presents the results from the hand-compiler HLSR2Soar agent. Soar's learning mechanism, chunking [27], is used to enable some self-optimization of performance with experience. The third and fourth row shows the results when the HLSR2Soar is run on the identical problem (third row) and a slightly different problem (fourth row), highlighting the difference in performance from the original HLSR2Soar (second row).

The performance metrics represent ways in which the performance of Soar systems is typically evaluated. These include total CPU time, the number of decision (perceive-decide-act) cycles, the total number of assertion cycles (within any decision cycle, Soar fully entails the current situation by iterative assertion cycles), total production firings, and total memory changes. Memory changes reflect the number of times an individual object (a Soar working memory element) is added or deleted from memory. This metric is important because a significant number of memory changes relative to decisions usually indicate performance problems.

	Produc-tions	CPU (msec)	Decision Cycles	Assertion Cycles	Prod Firings	Memory Changes
AMBR Soar Agent	27	140	7	15	23	120
HLSR2Soar	142 (RTL 54) (Com 49)	752	21	97	231	584
HLSR2Soar (repeat)	149 (chunks 7)	551	13	66	144	300
HLSR2Soar (different response)	151 (chunks 9)	651	18	82	176	420

Table 7: Total knowledge and performance comparisons for the feasibility demonstration. Note that RTL refers to runtime library productions, Com refers to compiled productions, and chunks refers to productions learned by Soar during execution.

The total number of productions in the HLSR2Soar agent was about five times greater than the base agent. The run-time library produces comprised about one-third of the total productions. This increase was surprisingly small; we expected about an order of magnitude increase in productions, due to the principle of rule specificity (see *Requirements for an HLSR Compiler*) and the overhead of translating HLSR into Soar (due to the limitations of the mappings). The results suggest that the initial mappings perhaps better leverage the architectural capability than we thought.

Total run-time also increased by a factor of about five, while decisions increased by a third, total productions by a factor of 6, and memory changes by about a factor of five. While the resulting HLSR2Soar agent is obviously slower, the overall performance differences are also not as great as expected, especially because we deliberately chose to simplify the initial micro-theories rather than elaborate them and increase complexity. Because memory changes scale (roughly) with production firings, this example suggests that HLSR2Soar agents will not become bogged down by memory changes. However, this example was probably of too limited a scale to understand fully the implications of the HLSR micro-theories on memory changes and overall expected performance.

Whenever any conflict occurs in the decisions of the HLSR2Soar agent, the micro-theories specify that a Soar impasse should be generated. Forcing the impasse leads to some initial performance costs, but allows the agent to reason about the conflict and bring any relevant knowledge to bear. Via Soar's learning mechanism, this approach also allows the agent to cache the results of its impasse deliberation. When the same task is repeated (third row), the agent realizes about a 25% speed up in performance. When the tasks are not identical, but similar, the speed up is more modest, 13%. However, these results are strongly encouraging, because they suggest that some of the performance costs of HLSR can be offset via the architectures' learning mechanisms, resulting in some self-optimization with experience.

3.2.6 Progress towards a prototype HLSR2Soar compiler

In addition to the hand-compilation feasibility demonstration described above, we also initiated the development of a rudimentary parser and code generation example, to ensure that we were not overlooking important problems in the automation of the compilation process. We implemented components of a Tcl-based (http://www.tcl.tk/) HLSR2Soar compiler. Soar is actually implemented as an extension of Tcl and there is already significant support for generating and parsing Soar productions implemented in Tcl. However, it is important to note that Tcl is only appropriate for rapid prototyping; the actual HLSR2Soar must be implemented in a scalable high-level language that has tools to support development and maintenance.

Tcl-HLSR emulates the syntax of HLSR as described in the HLSR language specification. HLSR keywords are implemented as Tcl commands in the same manner that Tcl keywords are implemented. The major difference between the Tcl-HLSR syntax and standard HLSR syntax is that code blocks are delimited with curly braces rather than the "end" keyword. This change simplified the development of the prototype considerably.

The Tcl-HLSR compiler recognizes keyword commands as they are sourced in the HLSR code file. The "action" of the command is to build a "parse tree" of HLSR objects. Tcl's upvar command is used to maintain a recursive parse context. When the code is completed, a parse tree of the HLSR code is represented in memory. This parse tree is then used as the basis for semantic analysis and code generation.

The code generation process involves traversing the parse tree and generating code as necessary. We developed a collection of Tcl procedures that simplifies the generation of Soar productions.

The implementation of the parser portion of the Tcl-HLSR compiler is mostly complete. The only constructs not covered by the parser are transforms and manipulator bodies. This incompleteness does not reflect a fundamental problem, but rather that this portion of the language was still being defined during the time the Tcl-HLSR compiler prototype was being developed. Furthermore, manipulator bodies will require "by-hand" parsing in Tcl, because nested calls to manipulators can't be encoded as Tcl commands. Given these complications, we did not attempt to implement manipulator bodies in the short period of time we had between their final definition and the end of the period of technical work for this effort. However, the prototype parser allowed us to demonstrate that the HLSR language (as defined in the grammar specification) could be parsed successfully and represented with a parse tree.

We did not make as much progress with code generation. However, we did demonstrate simple Soar code generation from the Tcl-HLSR-generated parse tree. We focused only on those elements of the language necessary for the AMBR speed request response example, as described above. The code generation prototype was sufficient for the activators, terminators, and some general language constructs from that example.

The implementation of the prototype, while very preliminary, does suggest that HLSR can be compiled, and that the code generation (at least to Soar) can be successfully automated. We also learned a number of technical lessons from the implementation, which will need to be addressed in more complete implementations of an HLSR compiler. Examples of these lessons include the need to define a convention for the naming of productions generated by the compiler, and the need to extend HLSR to make the unification of initializer and object more straightforward.

4 Results and Conclusions

4.1 Preliminary Evaluation

Our evaluation of HLSR to this point is preliminary. Further evaluations are necessary subsequent to the development of full scale compilers and large scale HLSR behavior models. The goal of this preliminary evaluation is to evaluate whether HLSR is capable of meeting its goals of enabling more efficient behavior development and enabling cross ISA behavior development.

The feasibility study described in section 3 provided significant insight into the feasibility of cross ISA compilation of behavior models. Here we provide three higher level evaluations. First, we describe how well HSLR aids the developer with the common development problems and solutions discussed in Appendix A. Second, we reflect on the requirements set forward in section 2.4 and the extent to which HLSR meets them. Third, we revisit the questions posed in section 2.1.1 and the tentative answers this effort has produced. While none of these evaluations can be considered conclusive, they do provide insight into the value of HLSR.

4.1.1 Solving Catalog Problems

To see how effectively HLSR helps alleviate some of the common problems and solutions that behavior developers must manage, we revisit the catalog described in section 2.2.1 and analyze how HLSR makes the developer's task easier. Table 8 provides a summary of the catalog problems, the potential solution provided by HLSR, and a brief description of any limitations of, or imposed by, the HLSR approach.

Problem/Pattern	HLSR Solutions	Issues not currently addressed by HLSR
Process tagging	HLSR provides several processes and constructs that reduce the effort a developer must spend for bookkeeping and tracking process tags. • Tagging process built into HLSR • Automatic tagging of goals by transforms • Automatic state tracking for goals and transforms • Process constraints provided in transform bodies	Automatic tagging is constrained to goals for now. Domain specific tags still require some developer management, but the built in tagging operation support makes this easier to encode and encapsulate.
Logic Tricks	HLSR provides logical OR and XOR operations. Furthermore, it prevents hard to maintain logical tricks by restricting what can appear in the left hand side of a production. Iteration blocks substitute for universally quantified variables.	HLSR does not have universal quantifiers because they do not have a good mapping to ACT-R. Also, HLSR requires more developer steps to encode complex logic; though the steps are much more maintainable than difficult to comprehend logic blocks.
Copying and Memory Manipulations	HLSR provides the **internalize** command to alleviate the need to create custom sensory system copying code. HLSR also abstracts away low level details like special cases when replacing one memory element with another.	HLSR does not yet support general copy processes in memory. General copy operations will be useful for planning processes, and should be added in the future.
Detailed Structure Specification	HLSR provides a high-level, ontological description of declarative knowledge. The details of how these structures decomposed and laid out in memory is the responsibility of the compiler rather than the developer.	HLSR still requires that declarative memory structures have type definitions, to allow for maintenance and reuse.

Planning	HLSR does not currently have built in support for the planning process. However, it does not prevent planning processes either.	HLSR still must address planning in future iterations. Likely components to be added are: • Improved support for general copying of memory structures • Support for execution commands "in the model's head" rather than externally • Support for ontological descriptions of process models used to do projections.
Writing production code procedurally	HLSR transform bodies provide constructs for encoding constrained processes procedurally. These map to production-based processes in Soar and ACT-R. Furthermore, HLSR greatly constrains the power of productions, thus limiting the hidden constraints and dependencies.	The interaction of transform body constraints and preferences is a new concept and has not been fully evaluated as to possible complexity it adds to development.
Knowledge Integration	HLSR will compile to an intermediate XML form, allowing integrated knowledge bases. Furthermore, HLSR itself can serve as a format for integrating knowledge across models and architectures.	HLSR only provides a first order solution. That is, it helps solve the technical format issues, but not semantic issues such as "what does the integrated knowledge mean?"
Implicit Semantics	HLSR provides many structures and processes for allowing and requiring domain-independent semantics to be made explicit. These include • Auto-tagging: make process tagging both explicit and automatic. • Type definitions: make structural constraints on declarative knowledge explicit • Process constraints in transform bodies: Make processing constraints and relationships explicit. • Activators and terminators: Make reactive consideration and reconsideration explicit.	The primary burden for understanding and encoding domain specific semantics is still on the developer. However, type definitions and more highly constrained language constructs make it easier for the developer to encode these semantics.

Goal Manipulation	HLSR provides a high-level representation of a goal that abstracts details about how a goal is structured or manipulated in the underlying ISA. The compiler rather than the developer decides the layout and interaction of goal data and tags.	The developer still is required to encode the high-level dependencies of a goal and to define the conditions under which a goal is met or has failed. This is because such information is often domain specific.
Perceptual Motor Interaction	HLSR provides a high-level abstraction for sensing and motor actions. This allows the compiler to encode consistent and, where necessary, theoretically sound perceptual/details for each sensing and motor function.	HLSR's representation may be too abstract. Part of the reason for this is that the ISAs themselves do not yet have generally accepted models of sensing and motor action. As the architectures begin to implement more sophisticated perceptual motor systems, HLSR will need to be updated to take advantage of the new theory and patterns that emerges.
Retrieve v. Compute	HLSR hides some of the details of retrieve vs. compute from the developer. This is true especially for learning, where the compiler is responsible for encoding knowledge so that it can take advantage of learning, and then deciding when to use learned knowledge and when to fall back on more general process knowledge.	HLSR does not have a good theoretical model of how the retrieve v. compute process should be decided and essentially leaves the hard work up to the compiler developer. Furthermore, HLSR does not address the retrieve v. compute issue for temporal knowledge (that is, do what was done in the past, or re-calculate it).
Output command structures & Proprioception	HLSR command objects are modeled after transforms and thus have a standard format for tracking output command state. The compiler and motor system are responsible for maintaining this information, not the developer. Furthermore, a few standard errors are supplied in the HLSR standard library to address common failures in command execution.	HLSR does not yet include solutions for the highest level semantics associated with commands. This includes being able to specify run-to-complete, start-stop, and run-while-present output semantics as described in Appendix A. It is not clear yet whether that level of control will be needed in HLSR.
List Management	HLSR provides a built in collection type that eliminates the need to build ad hoc list management structures in behavior models.	Ordered lists are included as part of the HLSR specification, but the mechanism for implementation may be awkward in practice. Further testing is required to be certain.

Iteration	HLSR provides a built in iteration construct that can be included in transform bodies.	Numeric-based iteration (e.g. for x = 1 to 10) is not directly supported, but is probably not really needed in most behavior models.
Understanding when a process involving multiple objects is complete	HLSR provides an iteration construct for implementing a search-based approach for this problem (see Appendix A).	Monitoring based solutions are not supported directly by HLSR, but HLSR does not prevent such solutions either. This is mainly because there are no known abstractions that encapsulate such a process, with the possible exception of aspect oriented programming [24].

Table 8: Catalog of common problems and solutions and the impact of HLSR on them.

We applied three different strategies for managing the problems posed in the catalog. First, we moved the responsibility for the problem from the developer to the compiler. Examples of this strategy include auto-tagging, sensory-motor interactions, and goal manipulation details. This approach has the advantages that the developer no longer has to think about the problem and the compiler can implement consistent, "best-practice" solutions to the problem. Unfortunately, this strategy removes all control from the developer and thus is not effective for problems where developers require some level of control over the solution.

Second, we implemented a solution pattern in HLSR, and left parameters for the developer to configure the aspects of the behavior that require domain knowledge or careful control. Examples of this strategy include the improved logic operators, iteration and list constructs, and transform body constraints. This approach has the advantage that many of the low-level details are abstracted from the developer, while things the developer cares about configuring are left available. However, this approach gains its power by constraining the space of possible solutions to a problem to common solution patterns. If a solution requires something beyond the standard solution pattern, the developer must solve it without direct HLSR support.

Finally, we provided the developer with a set of constructs and processes that are useful for solving the problems, but did not constrain the solution in any particular way. Examples of this strategy include the tagging processes and constructs, type definitions for structuring and encapsulating parts of solutions, and iteration processes for determining when all objects of a type have been processed. Even planning is supported to some extent using this strategy since planning processes can take advantage of all of the structure provided by HLSR to better implement the solution. This approach has the advantage that the developer can build a solution that best fits the details of the problem. HLSR still provides some support and makes the task easier than it would have been in native ISA code. However, this solution leaves the solution highly unconstrained. Thus the developer must understand the solution space well, manage the details of the solution, and know how to map HLSR constructs to the solution. Overall developer effort is highest.

In conclusion, we see that HLSR makes many of the tasks that inhibit efficient behavior development easier. However, HLSR does not solve all of these problems, and it is probable that HLSR by itself cannot solve all of them. We are convinced that some of the problems encountered in behavior development are best mitigated by improved architectural processes such as learning.

The process of building HLSR models will lead to other patterns and solutions common to HLSR development. These patterns and solutions will be at a higher level of abstraction than the low level issues discussed here, but are likely to be the source of further efforts to abstract the behavior development process. This would reflect the general progress of software engineering over the past decades where each level of abstraction is followed by another level that leads further toward rapid development.

4.1.2 Meeting HLSR Requirements

As part of our analysis we evaluated how well HLSR meets the requirements described in section 2.4. Our methodology was to score HLSR for each of the six core requirements on a scale from zero (does not meet requirement at all) to five (completely meets the requirements). Three of the HLSR team measures scored HLSR against its requirements in this way, and their scores were averaged to form a final score.

Figure 31 gives a qualitative picture of final results of this evaluation. The colored regions indicate how well we felt HLSR met each requirement. A pie slice that is completely shaded with color indicates a requirement that was completely met. A pie slice with no color shading indicates a requirement that was not met at all.

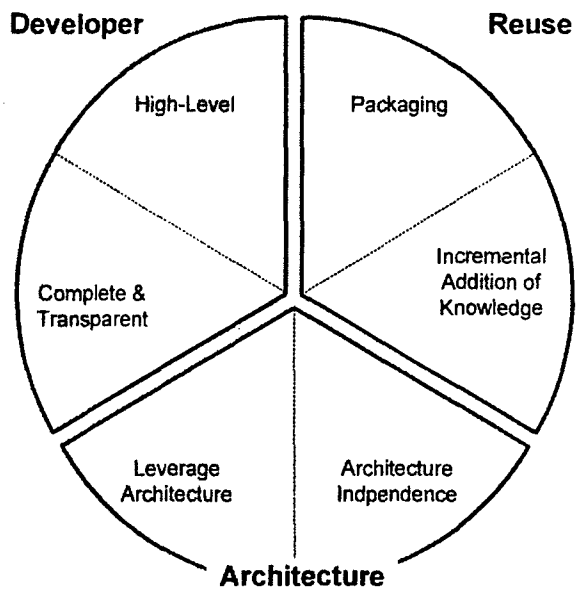


Figure 31: Results of evaluation of HLSR against its requirements.

Not surprisingly, based on our assertions in section 2.4, HLSR does not meet any requirement completely. This is because the requirements conflict, and to achieve all of these requirements simultaneously is impossible. Our goal was to achieve a perfect balance across all

requirements. However, our analysis indicates that HLSR currently fulfills the requirement to be complete and transparent at the expense of leveraging the unique capabilities of the ISAs. That is, one of the mechanisms used to obtain completeness, was to take some of the behavior control away from the architecture and give it to the developer. This tradeoff was not complete; however, as the HLSR specification still leaves a significant amount of power to the architecture through architecture discretion as discussed in section 0.

In other areas, we feel HLSR has achieved balance between the requirements. In particular, we all agreed that HLSR provides improved high-level representations, better packaging, and better support for incremental addition of knowledge than native ISA knowledge. We also achieved the important goal of being architecture independent. No construct or constraint in HLSR depends on an underlying ISA construct or process. However, HLSR is biased toward cognitive architectures over other architectures and would probably require some design changes to be applied to other types of ISAs.

Obviously, this evaluation was not scientific or statistically significant. The evaluations are clearly subjective and biased to some extent. However, the evaluators were all experienced ISA developers and felt collectively that HLSR has made significant progress toward higher-level behavior development. However, there will be a need for more formal and robust evaluations in the future.

4.1.3 Answers to Research Questions

In this section we revisit the questions posed in section 2.1.1 and consider them in light of the progress made in this effort. The answer to the fundamental question of cross-architecture compatibility is a qualified yes. The feasibility demonstration shows that a higher-level abstraction can be compiled to both Soar and ACT-R. However, ACT-R and Soar are actually

very similar architectures and the problem we chose did not fully exercise all aspects of the architectures (or of HLSR). Thus, we do not yet know for certain if HLSR will span both Soar and ACT-R in total and we do not know how well HLSR will generalize to other ISAs. By looking at HLSR as a distinct architecture, we can easily conclude that HLSR is compilable to any ISA, given a sufficient mapping and well-developed micro-theories. The still open question is the extent to which HLSR can leverage different underlying ISAs using the same language primitives (i.e., CCRU for beliefs, goals, transforms, etc.)

As Section 4.1.2 summarizes, the project team was not entirely satisfied with the extent to which HLSR leverages the underlying architecture capabilities. Dissatisfaction arises from the initial emphasis on functionality over the maximization of ISA leveraging and because each architecture supports different features more effectively than the other. Because the HLSR specification is informed by both architectures, it necessarily includes features that do not map as readily to one of the two architectures. For example, goals map directly into ACT-R structures, but do not map into any comparable Soar structure. Thus, HLSR goals do not leverage Soar's built-in capabilities as effectively as they do those of ACT-R. Finally, HLSR constructs may be inappropriately or improperly defined in this draft version of the language. The initial design was biased toward making development easier, rather than leveraging architecture capabilities. Future work on the compilation process will likely result in "push back" from the architectures, such that the language does better leverage the underlying ISAs.

This effort reinforced and clarified our early notions that ISAs are, conceptually, quite similar. Importantly, high level designs for these systems also share many similarities. The differences appear only in the details. The observed similarities depend to some extent on the

close similarities of Soar and ACT-R. However, beliefs, desires, and intention (BDI) systems [7, 49] share many of the characteristics of these architectures as well, suggesting a common design and language approach across the major agent-based approaches is perhaps within reach.

Commonalities and differences between ACT-R and Soar are presented in section 2.3. The commonalities led directly to many of the HLSR constructs as described in section 2.5, including the core constructs of goals, transforms, productions, and preferences (section 2.5.1). Thus the question "can abstractions be designed that take advantage of similarities in ISAs?" was answered affirmatively.

The principle of architecture discretion defines how we abstract and/or hide architectural differences. However, as is discussed in 4.4, remains to be explored the extent to which architecture discretion will ultimately influence behavior and whether that architecture discretion will be acceptable to developers.

We next address questions of high level programmability. Our analysis described in 4.1.2 indicates that HLSR is complete enough to encode behavior without the need to interact with the underlying ISA. HLSR has sufficiently powerful constructs to generate goal driven behavior, reactive behavior, and sensory/motor interaction. It has a memory model capable of representing any arbitrary structure. Its logic operators, both Boolean and predicate, are sufficient for implementing all of the standard logic queries.

HLSR's completeness begs the question of whether it is high level. In comparison, to Soar and ACT-R, it is. Almost every HLSR construct encapsulates multiple underlying ISA constructs. The runtime libraries required to implement portions of indicates that HLSR is not just a new interface to existing ISA features, but rather a new abstraction. Because HLSR solves or makes easier many of the common problems and solutions that developers typically encounter

in the details of implementing ISA behavior programs (see 0), these solutions suggest a higher level of abstraction.

Whether HLSR knowledge can be merged with learned knowledge remains an open question. As was presented in Section 3.2.5, it is possible to take advantage of the learning capabilities of the underlying ISA when compiling and executing HLSR knowledge. In fact, the compile process may make it easier to leverage the learning capabilities of the underlying ISA by allow ISA experts to tune the compile process such that it emphasizes the structure and procedures necessary for learning. This could lead to behavior models that learn robustly, even when less experienced developers build the models. However, the critical issue of mapping learned ISA knowledge back into HLSR was not explored in this effort. Because this problem has proven to be a fundamental issue in user adoption for previous attempts to build high level languages for ISAs [54], addressing this issue will be necessary for the success of HLSR.

The core HLSR constructs defined in 2.5.1 each encapsulates an important aspect of behavior. Type definitions encapsulate declarative knowledge structure. Goals, transforms, and preferences encapsulate the important elements of goal-driven behavior. Most importantly, object-oriented and procedural interfaces provide well defined access to and from these structures. These encapsulation constructs were not chosen arbitrarily, but were based on our research into the core capabilities of ISAs and on the design processes used to build ISAs. Therefore, we are confident that these constructs will be appropriate for constructing behavior models.

Effective reuse with HLSR is an open question. Reuse depends on many factors including:

- The ability of other developers to understand the component

- The ability to encapsulate the internal details of the component
- The ability to effectively expose the aspects of the component that may vary for different use cases
- The similarity of the use cases across which reuse is being attempted

For all but the last factor, HLSR provides significant improvements over native ISA representations. We feel HLSR is easier to understand (e.g. compare the compiled code to the HLSR code in the appendices). HLSR provides the ability to encapsulate the details of components into discrete units, and HLSR provides interfaces for facilitating the integration of components with larger models. However, HLSR cannot address the last factor. If the domains of reuse are different enough, and the goals and tasks are not similar across use cases, then very little reuse can be achieved.

HLSR provides some of the core primitive components required for reuse. These include goals, transforms, and interfaces. The most important of these components is interfaces. Interfaces are the abstraction behind which coupling hides. That is, interfaces make it easier to extract and adapt components to different models by isolating the decisions about which behavior units depend on other behavior units to well defined locations in the model. This facilitates the least commitment to dependency as described in section 2.2.2.1. Second, we believe that there are likely to be higher level encapsulation units that HLSR does not yet support. Among these are heterogeneous units that encapsulate a collection of goals, transforms, preferences and other constructs and form a *behavior unit*. Also among these are *behavior templates* which would allow a richer set of parameterizations on behavior, thus allowing behavior units to adapt to a broader range of models [1, 9, 20, 45].

Simple approaches to encapsulation conflict with pattern matching and learning. Pattern matching is used mainly for reactive behavior such as that provided by activators (section 2.3.3.4). It also plays a role in context driven conflict resolution such as that facilitated by preferences (section 2.3.2.4). In general it is very difficult to encapsulate knowledge representations such as productions because they have the potential to reference any memory element. ISA memory is intentionally designed to be fairly unstructured and globally accessible to ensure any context available for decision making is available. However, this potentially conflicts with encapsulation. HLSR query constructs (section 2.5.1.3) provide a buffer between the details of memory access and the constructs that access memory, but at some point the detailed mappings must be made between a query and what memory elements it references. Queries help to isolate those mappings, which are often volatile in that they change for every component use case. To our knowledge, this two-stage method of encapsulation for ISAs is novel and has the potential to make more strongly encapsulated modules feasible in ISAs, without introducing design-time restrictions which limits run-time flexibility. Whether queries actually improve modularity remains to be evaluated.

Learned knowledge is also difficult to encapsulate. Current ISA learning algorithms tend to be "bottom up" learning processes. That is they learn small pieces of knowledge that over time form bigger collections of knowledge that modify behavior. There is no built in support for constructing higher-level knowledge constructs out of the smaller learned knowledge pieces. Thus, it is almost impossible to identify the higher-level behavior units within learned knowledge unless the learning process is very carefully managed by the developer.

Finally, we address questions related to compilation process and efficiency. We explored two questions: the correctness and tractability of compilation. We have shown that HLSR

constructs have implementations in both Soar and ACT-R. However, we have not yet constructed a complete, automated compiler for this task. The prototype compiles a limited subset of HLSR into Soar, but it does not compile the most complex HLSR constructs such as transform bodies. However, most of the HLSR constructs and constraints appear that they can be automatically compiled straightforwardly.

However, for each ISA, a small subset of HLSR constructs and constraints may be difficult to automatically compile. For example, in ACT-R each declarative knowledge structure must be decomposed into "chunks" of limited size to allow spreading activation to function effectively. The process of decomposing declarative knowledge structures into smaller sized chunks is typically done by a developer. How appropriately this process can be automated is unclear.

We did not discover any theoretically intractable problems to compilation. Most HLSR constructs are not so far removed from the underlying ISA constructs that they require complex translations. It is possible that intractable problems do exist but have not yet been uncovered. The most likely candidate for such a problem is the problem of mapping learning back into HLSR code. Though we have not yet explored this problem sufficiently to make conclusions, this problem generally requires searching over arbitrary networks of interdependencies in search of higher-level patterns. This requirement for arbitrary search suggests the problem might be intractable, at least under current HLSR and ISA constraints.

These answers to the research questions of the project are based on sound support from the research and the small scale feasibility demonstration described in this final report. Detailed and more objective answers to these questions must wait for a more complete implementation and deployment of compilers for both Soar and ACT-R.

4.2 Accomplishments Summarized

This effort had both significant research and design components. We summarize our accomplishments in both, emphasizing results that can impact further work on HLSR and the field of behavior development in general.

4.2.1 Research Accomplishments

We analyzed and documented the similarities and differences between ISAs. Our primary focus was on Soar and ACT-R; however, concepts from other approaches, especially BDI systems were incorporated into our research results. From this analysis we were able to leverage architecture similarities to form architecture independent abstractions – the foundation upon which HLSR rests.

We analyzed and documented common problems and solutions encountered by behavior developers. The results are summarized in Appendix A. This catalog served as a driver for HLSR constructs and constraints, as well as a metric against which we evaluated the usefulness of HLSR (section 0).

We explicitly defined several developer and architecture principles (sections 2.2.2, 2.3.3, and 2.3.4) that serve as guidelines to behavior modeling and the foundations of many HLSR constructs and constraints.

4.2.2 Language Design Accomplishments

We completed a formal specification for HLSR. This specification describes the primitive constructs and constraints of HLSR in sufficient detail for compiler implementation and HLSR development.

The HLSR language design provides solutions for a large subset of the catalog problems and solutions as described in section 0. This implies that HLSR can help to solve real problems and make behavior development easier and more efficient.

The HLSR language is architecture independent. That is, it does not depend on any structure, constraint, or process defined in either Soar or ACT-R. Architecture independence was one of the explicit goals of this effort.

4.2.3 Compiler Design Accomplishments

We completed an initial mapping of HLSR constructs and constraints to both Soar and ACT-R. This mapping includes an ontological model of Soar, and a set of micro-theories that define how HLSR constructs and constraints should be interpreted within Soar. The ACT-R mappings are less comprehensive and in-depth than the Soar mappings (intentionally based on the statement of work), but are sufficient to show how HLSR knowledge can execute in ACT-R.

We demonstrated the feasibility of compiling HLSR to Soar and ACT-R through a hand-compiled example problem. This problem included a design, HLSR knowledge that implemented a slice of that design, and Soar and ACT-R knowledge resulting from the hand compilation process on the HLSR knowledge. The Soar knowledge was sufficient to be executed.

We created a prototype compiler that compiles a limited subset of HLSR to Soar. This prototype is not complete, but shows the feasibility of automatically compiling some of the core elements of HLSR – in particular activators and goals.

4.3 Lessons Learned

We observed several important lessons during this effort specific to both intelligent system architectures and behavior development:

1. ISAs have much more in common than expected. Though emphasis and low-level details are different between architectures, the general structure and constructs used to encode behavior tend are very similar (see 2.3). The development process for each ISA is also similar. Additional work that crosses architecture boundaries would benefit the behavior modeling community in general, resulting in the clarification of aspects of behavior modeling fundamental to intelligence. Furthermore, cross-architectural understanding will likely lead to ISA improvements in areas that are poorly supported in individual architectures. The evolution of the EASE architecture (a hybrid of ACT-R, Soar, and EPIC) typifies this cross-fertilization and hybridization [10-13].
2. Considering behavior at the HLSR level causes architecture "quirks" to be highlighted. That is, once we began to think at the level of HLSR, we found it no longer acceptable to think about the details, workarounds, and hacks that typically make up significant portions of native ISA behavior models (e.g., those detailed in the catalog of common problems). This change in perception suggests HLSR is both high level and useful and that HLSR would have an immediate benefit if fully implemented.
3. There is an inherent tension between high level representation and leveraging a specific architecture. ISA neutral abstractions necessarily do not leverage ISAs to the extent that architecture specific abstractions would. Thus, it remains to be

seen if the cross-architectural focus of HLSR is essentially the correct one. On the positive side, the cross-architectural focus identified representational problems common to two architectures. However, unless HLSR can be made to successfully leverage the full capabilities of the underlying ISAs, it is questionable if it will prove more useful than architecture-specific approaches.

4. Improvements to learning algorithms will probably be needed for efficiency and knowledge integration. The feasibility demonstration suggests that HLSR will potentially increase the size of a behavior model about five to ten times. Also, since native code is no longer hand tuned for specific cases and domains, some of the processing is more deliberate and slower in order to maintain correctness. Compiler optimization will inevitably solve some of these issues. However, the built in adaptation and optimization processes of ISAs may provide an even bigger factor for optimization. Learning can enable performance speed up, and can hopefully allow us to better integrate knowledge bases by optimizing access to this knowledge. However, current learning algorithms (at least in ACT-R and Soar) cannot provide this optimization robustly. Thus, pressure for self-optimization may encourage or speed the development of more robust approaches to learning.

5. It is possible to balance engineering and architecture requirements. HLSR indicates it is possible to constrain and encapsulate behavior models in a way that makes them more understandable and maintainable. This encapsulation can be realized without sacrificing the runtime flexibility of ISAs. The open question is the extent to which the abstractions in HLSR scale to large systems.

4.4 Key Open Issues

This effort answered many questions, but also introduced new questions and areas of further development.

The HLSR specification covers a broad range of behavior; however, some advanced features were not included in the specification due both to schedule constraints and the desire to ensure the functionality of basic features. The most notable missing element from the HLSR specification is the HLSR Standard Library. Though all of the language constructs and constraints are defined, it remains to specify the details of the standard reusable components of the language. The process of building this library will include the HLSR specification of these reusable components and the runtime library components that would be necessary to support these components in the ISA. The number and types of components that need to be specified include meta-reasoning type definitions and processes, standard failure objects, and standard command objects. The effort to define and implement the HLSR Standard Library is likely to be nearly as significant as building the initial specification itself, but is necessary for HLSR to be broadly useful in the development of behavior systems.

HLSR also still lacks some of the large scale packaging and template constructs that would make it easier to package large collections of constructs into behavior units. Future work on HLSR will seek to add these, and thus further improve the level of abstraction and support for reuse provided by HLSR.

Though HLSR was heavily informed by analysis of both ISAs and the development process, its constructs and constraints are biased toward making the development process easier. Behavior developers are the primary user of HLSR, and optimizing the language to make it and effective modeling tool makes it more likely HLSR will be used. Further, some of the ISA

issues and constraints cannot be fully understood until a fully functional compiler is developed. Our intention is to use the compiler development process as an opportunity to revisit some of the constraints and constructs in HLSR and modify them in ways that lead to better leveraging of the ISAs.

The most significant future development work is the implementation of a fully functional compiler for both Soar and ACT-R. This process will undoubtedly lead to new questions and lead to improvements in HLSR as discussed above, but more importantly, it enable the use of HLSR as a behavior programming tool. Once compilers have been constructed we intend to run experiments and cases studies to better evaluate the language and its impact on both development and the behavior of HLSR models.

Part of the process of compiler development will need to include the development of support tools, especially a debugger. Previous efforts at developing higher level behavior representations have had significant trouble being accepted because they lacked sufficient debugging tools [54].

The relationship between HLSR and ISAs also requires further elaboration. The integration of learning processes with HLSR models must be addressed both from the perspective of optimization, and understanding learned knowledge at the level of HLSR (see 4.1.3). HLSR may enable larger scale systems than current ISAs can handle. While ACT-R and Soar do not have theoretical bounds on the knowledge they can process, the implementations of these ISAs are not always capable of handling very large sets of knowledge (sometimes due to interactions between the virtual machine software architecture and the limitations of the physical architecture, such as when operating system paging is necessary for the consideration of an agent's complete knowledge base). Thus, HLSR will probably drive performance-based

refinements to ISA implementations. HLSR systems may also put pressure on ISA designers to implement portions of HLSR that are not well implemented in each ISA. Ideally, this process will lead to ISAs that are more complete and effective at generating intelligent behavior. It may also lead to the creation of a new ISA that integrates the best features of existing ISAs, or at least those features that best support HLSR.

Finally, we do not yet know if HLSR will scale to the really large models that HLSR was designed to support. We believe the constructs we have included in HLSR show promise of scaling well, but there is no formal way to prove this. The only proof is by construction; that is, by building such a system. Obviously building such models is dependent on having functional compilers.

5 References

[1] Alexandrescu, A., *Modern C++ Design: Generic Programming and Design Patterns.* 2001, Boston, MA: Addison-Wesley Professional.

[2] Anderson, J. and C. Lebiere, *The Atomic Components of Thought.* 1998: Lawrence Erlbaum.

[3] Anderson, J.R., D. Bothell, M.D. Byrne, S. Douglass, C. Lebiere, and Y. Qin, *An integrated theory of the mind.* Psychological Review, (in press).

[4] Aschenbrenner, P. and A. Schürr. *Generating Interactive Animations from Visual Specifications.* in *2003 IEEE Symposium on Visual Languages and Formal Methods.* 2003. Auckland, NZ.

[5] Beard, J., P. Nielsen, and J. Kiessel. *Self-Aware Synthetic Forces: Improved Robustness Through Qualitative Reasoning.* in *Proceedings of 2002 Interservice/Industry Training Simulation and Education Conference.* 2002. Orlando, FL.

[6] Best, B. and C. Lebiere. *Teamwork, Communication, and Planning in ACT-R Agents Engaging in Urban Combat in Virtual Environments.* in *2003 IJCAI Workshop on Cognitive Modeling of Agents and Multi-Agent Interactions.* 2003. Acapulco, Mexico.

[7] Bratman, M., *Intentions, Plans, and Practical Reason.* 1987, Cambridge, MA: Harvard University Press.

[8] Card, S., T. Moran, and A. Newell, *The psychology of human-computer interaction.* 1983, Hillsdale, NJ: Lawrence Erlbaum.

[9] Chandrasekaran, B., *Generic tasks in knowledge-based reasoning: High-level building blocks for expert systems design.* IEEE Expert, 1986. **1**(3): p. 23-30.

[10] Chong, R.S. and R.E. Wray, *Constraints on Architectural Models: Elements of ACT-R, Soar and EPIC in Human Learning and Performance*, in *Modeling Human Behavior with Integrated Cognitive Architectures: Comparison, Evaluation, and Validation*, K. Gluck and R. Pew, Editors. to appear.

[11] Cooper, R., *Modelling High-Level Cognitive Processes.* 2002: Lawrence Erlbaum Associates.

[12] Eisenberg, M., *End-user Programming*, in *Handbook of Human-Computer Interaction*, M. Helander, T.K. Landauer, and P. Prabhu, Editors. 1997, Elsevier Science B.V. p. 1127-1146.

[13] Fineberg, M.L., *A Comprehensive Taxonomy of Human Behaviors for Synthetic Forces.* 1995, Institute for Defense Analyses: Alexandria, VA.

[14] Fodor, J., *The modularity of mind.* 1983, Cambridge, MA: MIT Press.

[15] Forgy, C.L., *RETE: A fast algorithm for many pattern/many object pattern matching problem.* Artificial Intelligence, 1982. **19**: p. 17-37.

[16] Freed, M.A. and R.W. Remington. *Making human-machine system simulation a practical engineering tool: An Apex overview.* in *The 2000 International Conference on Cognitive Modeling.* 2000.

[17] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* 1994, Boston, MA: Addison-Wesley.

[18] Gluck, K. and R. Pew, eds. *Modeling Human Behavior with Integrated Cognitive Architectures: Comparison, Evaluation, and Validation.* to appear.

[19] Gray, W.D., B.E. John, and M.E. Atwood, *Project Ernestine: Validating a GOMS analysis for predicting and explaining real-world performance.* Human-Computer Interaction, 1993. **8**(3): p. 237-309.

[20] Hübscher, R. *Composing Complex Behavior from Simple Visual Descriptions.* in *IEEE Symposium on Visual Languages.* 1996. Boulder, CO.

[21] John, B.E., K. Prevas, D.D. Salvucci, and K.T.a.i.C. Koedinger. *Predictive human performance modeling made easy.* in *CHI 2004 Conference.* 2004. Vienna, Austria.

[22] Jones, R.M., J.E. Laird, P.E. Nielsen, K.J. Coulter, P.G. Kenny, and F.V. Koss, *Automated Intelligent Pilots for Combat Flight Simulation.* AI Magazine, 1999. **20**(1): p. 27-42.

[23] Jones, R.M. and R.E. Wray. *Comparative Analysis of Frameworks for Knowledge-Intensive Intelligent Agents.* in *AAAI Fall Symposium Series on Achieving Human-level Intelligence through Integrated Systems and Research.* 2004. Alexandria, VA: AAAI Press.

[24] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming.* in *European Conference on Object-Oriented Programming (ECOOP).* 1997. Finland: Springer-Verlag.

[25] Laird, J.E. and A. Newell, *A universal weak method,* in *The Soar Papers: Research on Integrated Intelligence*, P.S. Rosenbloom, J.E. Laird, and A. Newell, Editors. 1993, MIT Press: Cambridge, MA. p. 245-292.

[26] Laird, J.E., A. Newell, and P.S. Rosenbloom, *Soar: An architecture for general intelligence.* Artificial Intelligence, 1987. **33**(3): p. 1-64.

[27] Laird, J.E., P.S. Rosenbloom, and A. Newell, *Chunking in Soar: The anatomy of a general learning mechanism.* Machine Learning, 1986. **1**(1): p. 11-46.

[28] Lallement, Y. and B.E. John. *Cognitive architecture and modeling idiom: A model of the Wicken's task.* in *Twentieth Annual Conference of the Cognitive Science Society.* 1998. Madison, Wisconsin.

[29] Lebiere, C., *Constrained functionality: Application of the ACT-R cognitive architecture to the AMBR modeling comparison,* in *Modeling Human Behavior with Integrated Cognitive Architectures: Comparison, Evaluation, and Validation*, K. Gluck and R. Pew, Editors. (in preparation), Erlbaum: Mahweh, NJ.

[30] Newell, A., *Unified Theories of Cognition.* 1990, Cambridge, Massachusetts: Harvard University Press.

[31] Nielsen, P., J. Beard, J. Kiessel, and J. Beisaw. *Robust Behavior Modeling.* in *11th Computer Generated Forces Conference.* 2002.

[32] Noy, N.F. and D.L. McGuinness, *Ontology Development 101: A Guide to Creating Your First Ontology.* 2001, Stanford Knowledge Systems Laboratory: Stanford, CA.

[33] Padgham, L. and M. Winikoff, *Developing Intelligent Agent Systems: A Practical Guide.* 2004, New York: John Wiley & Sons. 230.

[34] Pearson, D.J. and J.E. Laird. *Example-driven diagrammatic tools for rapid knowledge acquisition.* in *Visualizing Information in Knowledge Engineering Workshop, Knowledge Capture 2003.* 2003. Sanibel Island, FL.

[35] Pew, R., K. Gluck, M. Young, R. Chong, and R. Wray. *The AMBR Model Comparison Project.* in *Cognitive Science.* 2002. Fairfax, VA.

[36] Repenning, A. and T. Sumner, *Agentsheets: A Medium for Creating Domain-Oriented Visual Languages.* Computer, 1995. **28**(March): p. 17-25.
[37] Rosenbloom, P.S. and J. Aasman. *Knowledge level and inductive uses of chunking.* in *Eighth National Conference on Artificial Intelligence.* 1990: AAAI Press.
[38] Rosenbloom, P.S., J.E. Laird, and A. Newell, eds. *The Soar Papers: Research on Integrated Intelligence.* 1993, MIT Press: Cambridge, MA.
[39] Salvucci, D.D. and F.J. Lee. *Simple cognitive modeling in a complex cognitive architecture.* in *Human Factors in Computing Systems: CHI 2003 Conference.* 2003: ACM Press.
[40] St.Amant, R. and F.E. Ritter. *Automated GOMS-to-ACT-R model generation.* in *International Conference on Cognitive Modeling.* 2004. Pittsburg, PA.
[41] Taatgen, N.A. and F.J. Lee, *Production Compilation: A simple mechanism to model Complex Skill Acquisition.* Human Factors, 2003. **45**(1): p. 61-76.
[42] Taylor, G. and R.E. Wray. *Behavior Design Patterns: Engineering Human Behavior Models.* in *2004 Behavioral Representation in Modeling and Simulation Conference.* 2004. Arlington, VA.
[43] Uschold, M. and M. Grüninger, *Ontologies: Principles, Methods and Applications.* Knowledge Engineering Review, 1996. **11**(2): p. 93-155.
[44] van Lent, M. and J.E. Laird. *Learning procedural knowledge through observation.* in *International Conference on Knowledge Capture.* 2001. Victoria, British Columbia, Canada: ACM Press.
[45] VanLehn, K., ed. *Architectures for Intelligence: 22nd Carnegie Mellon Symposium on Cognition.* 1991, LEA: Hillsdale, NJ.
[46] Wooldridge, M.J., *Reasoning about Rational Agents.* 2000, Cambridge, MA: MIT Press.
[47] Wray, R.E. and R.M. Jones. *Resolving Contentions between Initial and Learned Knowledge.* in *Proceedings of the 2001 International Conference on Artificial Intelligence.* 2001. Las Vegas, NV.
[48] Wray, R.E. and R.M. Jones, *An Introduction to Soar as an Agent Architecture*, in *Cognition and Multi-agent Interaction: From Cognitive Modeling to Social Simulation*, R. Sun, Editor. To appear, Cambridge University Press: Cambridge, UK.
[49] Wray, R.E., J.E. Laird, and R.M. Jones. *Compilation of Non-Contemporaneous Constraints.* in *Proceedings of the Thirteenth National Conference on Artificial Intelligence.* 1996. Portland, Oregon: AAAI Press.
[50] Wray, R.E., J.E. Laird, A. Nuxoll, D. Stokes, and A. Kerfoot. *Synthetic Adversaries for Urban Combat Training.* in *2004 Innovative Applications of Artificial Intelligence Conference.* 2004. San Jose, CA.
[51] Yost, G.R., *Acquiring Knowledge in Soar.* Intelligent Systems, 1993. **8**(3): p. 26-34.

6 List of Acronyms

ACT-R	Adaptive Control of Thought – Rational. Within this report, ACT-R refers to a hybrid cognitive architecture combining symbolic and sub-symbolic processes [2, 3].
AMBR	Agent-based Modeling and Behavior Representation
AOP	Agent oriented programming
ATC	Air traffic controller
BDI	Belief, desire, intention. BDI is a logic-based methodology for building competent agents, driven by the basic assumption that intelligent agents ought to be rational in a formal sense [7, 33, 46].
BNF	Backus-Naur form
CCRU	Consider, Commit, Reconsider, Unconsider – the four primitive state change operations that can be performed on any HLSR object.
G2A	A GOMS to ACT-R compiler
GOMS	Goals, Operators, Methods, and Selection Rules. GOMS is a methodology based in psychology and human-computer interaction that formalizes many details of high-level human reasoning and interaction [8, 16, 19].
HLSR	High-Level Symbolic Representation
HLSR2Soar	The initial HLSR-compiler designed and prototyped in the execution of this project
ISA	Intelligent System Architecture
KE	Knowledge engineer

OO	Object-Oriented
RTL	Runtime Library
SME	Subject matter expert
Soar	Not an acronym. Within this report, "Soar" refers to a cognitive architecture based on principles of functionalism and parsimony [23, 26, 48].
TAQL	Task acquisition language
WME	(Soar) Working memory element
XML	Extensible Markup Language

Appendix A : Complete Catalog of Problems and Solution Patterns

Catalog of Low-level Details HLSR Should Abstract

Process Tagging

An example of a low-level task is symbolic tagging. Symbolic tagging is the process of creating and managing information about the processing of one or more objects. A simple example of symbolic process tagging is marking a message as "sent" after processing it with a "SendMessage" transform. Symbolic tags are common and often scattered widely throughout the declarative knowledge of an agent and serve important functional purposes such as organizing processing information related to knowledge structures and serving as markers for process control. While functionally necessary for managing processes within the least commitment framework of cognitive architectures, tags tend to clutter agent knowledge with small details and repetitive processing.

What is needed is a way to better structure and automatically maintain these process tags such that they do not clutter the core processing code.

Logic Tricks

Soar and ACT-R only provide direct support for a fairly minimal set of logic operators and expressions. While functionally complete, these limited logic operators make answering questions like "are there any objects that have not been processed" as well as simple logical "OR" conditions very difficult to write and even harder to understand later during maintenance. This is a source of bugs in intelligent system programs.

What is needed is a more developer-friendly set of logical operations that can be mapped to the more complex logic required for architecture implementation.

Copying and Memory Manipulations

Two common and often painful low-level details in Soar is creating a declarative structure by copy and populating one to many, and many to many elements of structures. This process is common, especially when internalizing sensory information. Solutions to this problem follow the same pattern, but require hand written and maintained copy code for each type of structure being copied.

In addition, Soar constrains memory such that if an attempt is made to replace a working memory element (WME) with a WME with the same values using a single production, the WME is actually removed. While functionally consistent with architecture design principles this constraint leads to the need to write two productions where, conceptually, one should suffice. Multiple productions that do the same thing are a source of maintenance errors.

What is needed is to provide convenient standard processes for copying and maintaining memory structures such as primitives for copying and abstraction of detailed memory constraints.

Detailed structures specification

The first step in specifying an ACT-R model is the definition of chunk types that will define which goals can be solved and which chunks can be represented in declarative memory (incidentally, it is usually considered good form if those are one and the same, but correspondence is usually imperfect at best). This requires defining for each type its slots, and

possibly its parent type (and inherited slots), taking advantage of a single inheritance mechanism. While in principle there is nothing wrong with that practice, which corresponds to defining the data structures in a typical computer program, the theoretical restrictions on memory access and theory makes that specification a very painful, iterative and time-wasting business with highly dubious cognitive validity.

Goal structures are defined with a particular purpose in mind, and then slots need to be added to accommodate various information-processing needs, leading to structure bloat. Various manipulations are attempted to minimize the number of transitory slots, with the result that some slots often lose any semantic association and become merely structural bins, which is contrary to their theoretical interpretation. Then chunk structures are further manipulated and duplicated to provide the other side of functionality, i.e. associative access to information. Often, the strong restrictions on memory access (sequential, non-backtracking) leads to the insertion of various shortcuts, duplications and representational gerrymandering with little cognitive plausibility.

What is needed is a representation that abstracts from the hard structure-slot specification and allows knowledge associations to be expressed naturally, in their intuitive context (when they are created and/or accessed), and with no additional computational details.

Catalog of High-level Tasks HLSR Should Make Easier

Planning

Here, planning refers to general AI planning which typically involves the following steps:

1. Evaluate the current state of the world as the agent sees it
2. Consider one or more actions on the world, projecting the agent "imagination"
3. Evaluate the imagined state

4. Compare the evaluations of the current state to the imagined effects of the various actions that can be taken. Whichever has the best evaluation is typically chosen.

A planning process typically produces as an output, a list actions and time/decency constraints on those actions. Plans are executed when the planned actions are taken. In an unpredictable, interactive environment, it is necessary to replan from time to time to take into account unanticipated changes in the environment, and change the plan accordingly.
Planning is used in many intelligent system behavior models and in aspects of models that enable robust execution.

Here are some things typically done in planning:

- Creating and manipulating imaginary states containing a subset of memory relevant to the planning operation
- Projecting the impact of actions. This requires some model of the impact of an action.
- Evaluating and comparing states (imagined and real). This typically involves the use of a utility function and some math in traditional systems, though different systems may use different techniques.
- Queuing actions for future execution
- Evaluating actions when it is time to execute them
- Modifying a plan if it is no longer acceptable
- Interleaving planning with execution (if an agent always plans, it can never act)
- Remembering what has already been acted on and what actions have been taken to avoid "common sense errors" (e.g. undoing an action immediately after doing it).

It would be inappropriate to put any dependencies for a particular type of planning algorithm or process into HLSR; however, features to help with the above processes would make it easier to build planning systems.

Writing production code procedurally

In Soar and ACT-R, it is often necessary to write lots of little productions to do one conceptually simple task. Typically, these productions have dependencies, are very closely related, and do a single high-level task such as operator application or state elaboration (e.g. copy down relevant information from a higher level). Following the logic of these closely related productions is difficult because they have subtle dependencies.

HLSR can provide mechanism for specifying closely related productions, but in a way similar to procedures. The high-level task can be specified explicitly (e.g. execute this sequence of actions), and the highly dependent set of productions can be generated and maintained by the compiler.

Knowledge integration

Knowledge integration is the process of integrating knowledge from multiple sources. Knowledge integration is difficult, even within knowledge bases designed for the same architecture. The difficulty is due to insufficient encapsulation and modularization and low-level programming idioms that leverage the underlying architecture in fundamentally different ways. These problems are not simply poor programming practice, but rather are fundamental issues related to the core principles and processes of cognitive architectures. For example, architectures prefer unencapsulated knowledge because it can more easily be brought to bear in

any applicable context. Pervasive coupling tends to occur because of the complex dependencies required for intelligent behaviors.

HLSR can provide mechanisms for better managing and hiding the complexity of integration, including encapsulation mechanisms, interfaces, and abstraction of the architecture leveraging mechanism to ensure that components work properly together.

Implicit semantics

Consider the production below (requires knowledge of Soar):

```
sp expect-response*request-intent-known-for-contact*intent
   (state <s> ^operator <o>
              ^incoming-message <m>)
   (<o>       ^name classify-contact
              ^contact <c>)
   ( <m>      ^name request-intent-known-for-contact
              ^performative reply
              ^content [ << hostile friendly neutral >>
<intent>])
-->
   (contact <c> ^intent-known-by-partner t
                ^intent <intent>)
```

Basically, if `intent-known-by-partner` is `t` and `intent` is present, that implies the agent got intent information from its partner (`intent-known-by-partner won't be present otherwise`). Other examples of such implicit semantics include sets of productions intended to fire in sequence, but with no explicit structuring to indicate this, as well as implicitly defined declarative structures where the only way to understand the intended declarative structure is by analyzing a large set of productions that potentially interact with it. Examples of such implicit semantics are common in both Soar and ACT-R and makes using systems built by others very difficult.

HLSR should make it both possible and required (when appropriate) to declare these types of semantics as explicitly for both improved error checking and improved maintenance.

Goal Manipulation

Finding the right level of support for goal operations is difficult. In ACT-R up to ACT-R 4.0, a goal stack provided the obvious support for basic operations such as pushing and popping subgoals on the stack, with only the top goal being accessible at the current time. It was found to be undesirable for two reasons:

1. The goal stack offered a perfect memory for past goals, a cognitively implausible proposition
2. It restricted flexibility and reactivity in dynamic environments.

In ACT-R 5.0, the goal stack has been removed and users rely on general-purpose declarative memory to store and retrieve past goals. This works reasonably well though it can be brittle and require significant attention to detail, which often leads to some hacks which are not really any more plausible than the old goal stack.

What is needed is a high-level way of handling goals that provides the required flexibility to handle dynamic environments without overburdening the user with a complex, heavy-duty mechanism.

Perceptual/Motor Interaction

In cognitive architectures, procedural and declarative operations are usually the focus of attention and perceptual/motor operations are usually handled in an ad hoc fashion. Starting with the PM modules to ACT-R 4.0, further integrated in ACT-R 5.0, ACT-R started treating

perceptual and motor operations on an even level with goal and declarative manipulations (the procedural module is similar but plays a more central role). This had a number of advantages, including better reactivity and an attractive unification, but handling perceptual and motor operations at that level usually requires a high level of attention to low-level details that the user would usually rather not be concerned with because they are often not central to the task at hand. There is a general intuition that those details are taken care of cognitively in a much more automated fashion.

HLSR should provide a high-level solution that would inherit the accuracy of the architectural perceptual/motor modules without requiring dealing with their internal details.

Catalog of Micro-Patterns That Developers Typically Use

Retrieve vs. Compute

In general, this micro-pattern expresses the choice between retrieving a previously stored solution to a specific problem and recomputing the answer using general procedural means. Both Soar and ACT-R use mechanisms to automatically store a problem solution after it has been computed. In Soar, it results in the creation of a new production that links directly antecedents to conclusions. In ACT-R, reflecting that architecture's procedural-declarative distinction, the goal holding the solved problem becomes a declarative chunk of information in long-term memory. This distinction is reflected in the mechanisms for resolving the choice. In ACT-R, the selection between the more general procedural computation and the more efficient declarative retrieval is arbitrated through subsymbolic quantities attached to declarative instances (activation) and production rules (utilities). In Soar, the choice is resolved using symbolic preferences. While those mechanisms have successfully accounted for increased robustness and

efficiency through learning, they are somewhat difficult to automate and often require careful debugging to ensure that they operate correctly.

HLSR can improve the robustness of those mechanisms through two separate means. One is to rely on the compiler to analyze the learning process and infer correctly at compile time the details that human modelers often go through a trial-and-error process to derive. Another potential solution is to endow HLSR with meta-cognitive knowledge of its own operations to allow the underlying model to adjust any flaws in its learning at run time.

Output command structures & Proprioception

This would be an HLSR "micro-theory" of the annotations and processing of tags for outputs. Output processing is not very well defined at the architecture level and so the developer ends up re-creating a lot work from previous efforts.

For Soar, there are generally three types of output commands:

- run-to-complete (once initiated, the command runs until completion/failure)
- start-stop (once initiated, run until a stop command is issued)
- run-while-present (once initiated, run until command is removed from output link)

There are also different status levels, which generally apply only to run-to-complete (we haven't explored the other options as much)

- **complete** (command completed)
- **did not compute** (syntax error/malformed command)
- **pending** (command received but not yet executing)
- **executing** (command is executing)
- **failed** (command has failed)

Importantly, there is often explicit proprioceptive feedback associated with commands. Proprioceptive feedback in most Soar systems is provided on the output-link, rather than the input-link (and this is the only time in which it is acceptable for outside information to come into Soar anyplace other than the input-link). One reason for this exception is that matching output commands to structures on the input-link is hard. Feedback is both comparative (this is where I want to be, this is where I am) and proprioceptive (Actuators are engaged).

All of these details can be defined by HLSR, so the programmer only needs to identify that an output was some specific type and HLSR would do the rest. So HLSR would define the types of output commands and supported processing for each type, and the user would simply identify the output commands as instances of the supported types.

List management

Creating and managing lists in both Soar and ACT-R is repetitive and clumsy. It often requires the annotation of objects with links to next and/or previous elements, which clutters the object with data irrelevant to its core semantics. Furthermore, several productions are needed to manage the iteration process as there are no built in iteration processes in Soar and ACT-R.

HLSR can improve list management by providing basic container data structures and basic processes on those structures. The details of which productions are necessary for implementation can be left to the compiler.

Iteration

Frequently, agents must iterate over items, for example to count or calculate a total of all the items of some particular type. In Soar 8, this counting generally requires one operator instance

for every counted object. This is both inefficient (computationally) and tedious to program correctly. ACT-R requires similar solutions, but using its own architecture mechanisms.

HLSR should make it easy to count things, perhaps by including as a language something comparable to an iterator, which would then get expanded into Soar and ACT-R productions/chunks to actually perform the counting.

Understanding when a process involving multiple objects is complete

Describing this pattern is easiest with an example. Consider a situation where one agent has command and control of other agents and wishes to ensure that each entity has acknowledged its orders before beginning to execute a mission. As a developer, you might write something equivalent to:

```
If all entities have acknowledged their orders
then consider a goal to execute the mission
```

This is a typical instance of the general problem where an agent needs to know when every element of a group has been processed in some way. There are two large categories of solutions to this problem, with a third hybrid solution.

The first category of solutions involves searching after the process in question has occurred. For example, searching after all of the orders have been acknowledged. Mentally, the agent records what messages have been received, and then uses a pattern defined by the developer to search back over all of the entities of interest and determine if the process has completed on them all. This approach works well in Soar and is often used in traditional software systems as well.

The second category of solutions involves monitoring a process and gradually building up the information necessary to answer the overall question (have all agents acknowledged their orders?). This approach is most effective in the following circumstances:

- When you know in advance that you need to answer the question (before doing the process). In fact, this is required.
- When the search process is too time consuming or prone to failure (e.g. ACT-R)

There are several ways this can be implemented (in fact, their may be a theoretically infinite number of ways this can be implemented), but here are described three straightforward approaches.

The first approach is to insert into the processing of each element, code to count how many elements have been processed. For example, each time an acknowledgement is received from an entity, add to the count of agents that have acknowledged their orders. When the count reaches the number of agents that should be responding, then it is clear that all agents have responded.

The second monitoring approach is to create a collection that contains all of the objects that have been processed. Now, as each object is processed, move them from the already existing collection of objects that have not been processed (assuming all have not been processed at the beginning) into the collection that has been processed. When the collection that contains the non-processed objects is empty, then the process is complete.

The third approach only works for ordered collections (e.g. words in a sentence). Simply stated, it is continue processing until the end statement is found (e.g. a period). Then mark the collection as having been processed.

A common thread among the monitoring processes is that they require modification to the process that's occurring in order to work. In particular you need three things:

- A monitoring process that executes whenever one object is processed
- A tag to store the intermediate process information (and, of course, an object to tag -- it must have a lifetime longer than the process that is occurring).
- A trigger (e.g. production) that monitors the tag and fires whenever the information in the tag indicates that the process has occurred for all elements in the collection.

To be complete (i.e. to handle the case when you are unable to start monitoring on time), you would need a standalone process (probably an extra goal combined with the transform that is monitoring progress, and maybe some additional control logic) to essentially do a manual search. This manual search iterates over each element with the sole purpose of determining if they meet the processing criteria.

One interesting distinction between the search and monitor approaches is that the search approach is easy to specify declaratively, and that the monitor approach requires procedural code. A second distinction is that the search approach is more completely decoupled from the process being executed on the objects while the monitoring approach requires integration of two processes -- the core process such as handling and encoding a message, and the secondary process to monitor progress.

The HLSR solution is not completely clear. One approach might be to allow universally quantified variables. But this might not work well for ACT-R because of the limited expressive power of its productions. Another approach might be to provide some sort of iterative search

process that is the equivalent of a universal quantifier, but that is does not require as much production complexity.

Appendix B: Standard Behavior Primitives

Behavior primitives are basic, atomic functions that do not require thought but realize the interface between the cognitive layer and the physical/psychomotor/conceptual layers of activities. Fineberg [13] offers a number of such primitive operations at varying levels of abstraction.

Movement Commands

"Move" is a common behavior primitive used by any mobile agent. This primitive abstracts away the details of how an agent moves, and provides a high-level interface to the process.

The ways to describe movement are constrained only by the number of freedoms of the kinematics of the vehicle being operated. HLSR allows the underlying vehicle interface to define what these degrees of freedom are, but specifies the speed, direction, and rate of change for each degree.

For example, in a three degree of freedom flight system, the controls that may be varied are "speed," "heading," and "altitude."

Commands

Command	Parameters	Explanation
Set	controller, real-number, units	Make the value of the controller become the indicated number when measured in the indicated units
Increase	controller, real-number, units	Modify the value of the controller by the indicated number to increase (if positive) or decrease (if negative) the current amount of the controller when measured in the indicated units

SetRate	controller, positive real-number, units	Make the rate of change of the indicated controller be the indicated number when measured in the indicated units
Tolerance	controller, real-number, units	Indicates the value of the acceptable deviation, in the indicated units, of the controller.

Values

The following are examples of the values that may be used for the controller parameter for a vehicle. They are listed in order from most general to most specific. Except in very degenerate cases, a vehicle should be able to move in two dimensions by adjusting speed and heading.

- Speed
- Heading
- Altitude
- Depth
- Pitch
- Roll
- Yaw

Discussion

The movement command begins the movement process. The actual movement is asynchronous with the reasoning process. The move command provides no direct feedback, though the agent should notice a change in its position. The move continues until the value of the controller is within the indicated tolerance, and then it is maintained within that tolerance until another movement command is issued.

When the indicated value is zero, the vehicle should stop movement along that control dimension. There should be no variation for tolerance.

Movement rates and tolerances should default to reasonable (non-zero) values if not specified.

Move commands overwrite previous move commands. When a value is "set" the vehicle will adjust at the indicated rate until that value is achieved within the indicated tolerance. When a value is increased, it will be adjusted from the actual current value of that control dimension (not the currently set value) at the indicated rate until it is within the indicated tolerance.

Any number of move commands may be executed by an entity at once time, but the vehicle platform may impose additional constraints on the order in which they are carried out.

Failures

Controller failure: The indicated controller cannot be moved because of some internal damage or misconfiguration.

Unable to execute: If the underlying platform is unable to carry out the movement command for any reason (e.g., blocked)

Expectation violation: The agent should expect to see a variation in the controller within a short period of time. If this does not occur, the agent may reissue the command or take additional corrective actions.

Feedback

- The current actual value of the controller should be available to the agent at all times.

- The rate of change of the controller should be available to the agent at all times.
- The tolerance of the controller should be available to the agent at all times.

Communication Commands

"Communication" is used for interaction between agents or between humans and agents. This primitive abstracts away the details of how the channels of communication are formed, and provides a high-level interface for the communication process.

Commands

Command	Parameters	Explanation
Communicate	medium, message	Using the indicated medium transmit the indicated message
SetChannel	medium, channel	Adjust the channel that the medium is tuned to, if multiple channels are possible
Abort	medium	Cease further transmission of any uncompleted message on the indicated medium

Values

The following are examples of the values that may be used for the medium.

- Voice
- Radio
- Data link
- Telephone

Most devices, with the notable exception being voice, have multiple channels. For example a radio can be set to different frequencies and the telephone can be dialed to different phone numbers. The actual value of the channel will be dependent on the medium selected. Once set it will be used for subsequent communications on that medium until changed.

The message is a text string, with possible (optional) mark-up for speech synthesis.

Discussion

The actual communication is asynchronous with the reasoning process and may take some time for the communication to complete. The communication command provides feedback when the message is complete and indicates that the medium is busy while the message is being sent. Once transmission has begun, the message cannot be modified, though it may be aborted and a new message sent.

The channel should default to a reasonable value if not specified.

Communication commands do not overwrite previous communications. If a second communication is sent, while the medium is busy, the second message should be queued for transmission once the first message is complete.

The "abort" command will cause all current and queued messages on that medium to be terminated immediately.

Failures

Medium failure: The medium is damaged or misconfigured.

Feedback

- *Medium busy status*: An indication of whether the medium is currently in use.
- *Current channel*: The channel each medium is currently tuned to.

- *Message sent (medium, message)*: The medium has completed transmission of the indicated message. This message is optional.

Device Deployment Commands

Device deployment commands can be used to initiate a wide range of behaviors that are carried out by the underlying physical system once set in motion. For example, shooting a weapon, taking a picture, or pressing a button on a keyboard.

Commands

Command	Parameters	Explanation
Deploy	device	Activate the indicated device
Aim	device, direction	Modify the alignment of the indicated device to point in the indicated direction
AimAt	device, object	Modify the alignment of the indicated device to point at the indicated object
Load	device, integer number, units	Increase (or decrease) the number of resources of the indicated unit type

Discussion

The deploy command initiates a process. The actual process is asynchronous with the reasoning process, and once set into motion it cannot (typically) be undone.

The number of times this process can be repeated may be limited by the resource consumption of the underlying device, and there will be feedback on the number of available resources, as well as a command for loading more resources into the device.

The device may or not require aiming in order to be deployed, and may be deployed without aiming, but the results may be highly undesirable.

Failures

Jammed: device failed to deploy due to damage or misconfiguration.

Full: device cannot be loaded with further units

Feedback

- *Deployed*: Signals that the message to deploy was received by the device
- *Aimed*: Signals that the device is currently aimed at the requested object or in the requested direction.
- *Resource count*: The number of times the device may be deployed before further loading.
- *Aim direction*: The direction the device is currently pointing
- *Locked*: An optional value indicating that the device is pointing at an object.

Debugging Commands

Debugging commands are used to assist the task of software development. HLSR requires that two such commands be provided, though the underlying architecture is encouraged to provide more.

Commands

Command	Parameters	Explanation
Print to Screen	condition, message	When the condition becomes satisfied, display the indicated message on the user's screen
Interrupt	condition	When the condition becomes true, stop the process in such a way that it may continued.

Values

The conditions are arbitrary expressions that can be evaluated within the underlying architecture to determine whether they are satisfied.

The message is a parameterized text string.

Discussion

The same set of beliefs match the conditions should only trigger the indicated action once; however, subsequent new combinations of beliefs which also match the conditions should again trigger the indicated action.

- Failures
- None
- Feedback
- Message appears on screen or program halts.

Appendix C: Relevant HLSR Code for AMBR Example

```
# Message Type definition,  We inherit multiple versions for
different types
#  of messages
type Message

 # Internally defined type
 type ContentCollection isa collection of * content isa symbol

 # References expected for a message
 w references from isa Entity
 w references to isa Entity
 r references contents isa ContentCollection

 # Create a message with to and a from
 init Message(<f> isa Entity, <t> isa Entity)
  consider <self>.from <f>
  consider <self>.to <t>
 end

 # Access to the message
 interface IMessage

  # Add content to the message only through this interface
  manipulator AddContent(<c> isa symbol)
   require decide consider <self>.contents.content <c>
  end

 end

end

# Message Type definition,  We inherit multiple versions for
different types
#  of messages
type RequestSpeedChangeMessage isa Message

 # Expected terms in
 string enumeration SpeedRequestTokens
  "request"
```

```
   "speed"
   "change"
  end

  # Access to the message
  interface IRequestSpeedChangeMessage

   # Gets the speed from the content. Here we depend on the
 content
   #  being of the form "speed 500"; however anything could
 appear
   #  between speed and 500 (e.g. "speed change 500")
   query SpeedRequested(<speed> isa number)
    activated contents.content <speed>
    contents.IsBefore("speed", <speed>)
   end

 end

end

# blip
# This is just a copy of Jacob's blip declaration.  However, Im
assuming this
# is *not* a sensed type, since there are items here that should
be derived as
# well as sensed.'

type Contact isa Entity
   w references speed           isa number
   w references location        isa Location3D
   w references color           isa BlipColor
   w references id              isa number
   w references screen-location isa Location2D
end

# Propose a goal when speed request is present
activator ProposeGoalAnswerSpeedRequest
            (<message> isa RequestSpeedChangeMessage)
invariant
   # Must have a sender to instantiate the goal
   activated <message>
   activated <message>.from <f>
then
   consider <g> new AnswerSpeedRequest()
end
```

```
##########################
# Goal
goal AnswerSpeedRequest isa AchievementGoal

 # Local memory for goal
 r references msg isa RequestSpeedChangeMsg

 # Initializer
 init AnswerSpeedRequest (<msg> isa RequestSpeedChangeMsg)
  consider <self>.msg <msg>
 end

 # Standard interface for goals
 interface IGoal

  # Im saying this goal is met once we have committed
  #  to a reply and have sent it.
  query InternalIsMet(<reply> isa MessageReply)
   activated <self>.msg tagged <reply>
   activated <reply>.msg tagged MessageSent
  end

  # Can't easily detect failure without some sense
  #  of time.  Here is an attempt.  The idea is to
  #  believe the goal has failed when I have rejected
  #  a reply. This may or may not make sense depending
  #  on agent design.
  query InternalHasFailed(<r> isa string)
         past considered <self>.msg tagged <reply>
     not past activated <reply> tagged MessageSent
         unconsidered <self>.msg tagged <reply>
  end
end


##############################
# Stuff related to transform
string enumeration SpeedRequestResponseOptions
 accept,
 reject
end

type SpeedRequestResponse
 r references response isa SpeedRequestResponseOptions
end

type AcceptSpeedRequest isa RequestResponse
```

```
  init AcceptSpeedRequest()
   consider <self>.response "accept"
  end
end

type RejectSpeedRequest isa RequestResponse
  init RejectSpeedRequest()
   consider <self>.response "reject"
  end
end

# REW: 2004-04-29: Add activator for the transform
# Jacob's note:
# The special syntax "for <goal>" is used to consider transforms
which always
# bind to a goal.  However, you still need to say _which_ goal
on the LHS.  In
# this case Im saying to propose a SimpleAnswerDecision
transform for any
# AcceptRejectResponse goal.

# Propose to do execute a simple decision
activator ProposeTransformSimpleAnswerDecision
          (<goal> isa AnswerSpeedRequest)
invariant
    # Assuming no other conditions make sense
    activated <goal>
then
    consider new SimpleAnswerDecision for <goal>
end

# Transform to decide how to answer a speed request
transform SimpleAnswerDecision

 # references fragments from this interface
 references interface ISendMessage

 # The goal the transform is attempting to achieve
 goal isa AnswerSpeedRequest

 # Transform local memory
 references msg isa SpeedRequestMessage

 # This initializer takes a message. If the transform
 #  is created using a default initializer (one that
 #  does not take the parameter), then the elaboration
 #  below provides the message object.
```

```
init SimpleAnswserDecision(<m> isa SpeedRequestMessage)
 consider <self>.msg <m>
end

# Show how an elaboration can work if a default
#  initializer is used.  There is no magic, it is just
#  a simple copy of goal.msg to the local memory.
elaboration StoreMsg()
invariant
 activated <self>.goal.msg <m>
then
 consider <self>.msg <m>
end

# Core behavior
body(<r> isa SpeedRequestResponse)

 # These are not explicitly ordered, so the compiler is
 #  free to try to implement the required statement first
 #  if it likes.

 # Decide the response we want to have
 ChooseResponse()

 # Bind to the choice that was activated.
 activated <self>.msg tagged <r>

 # Now propose a goal to actually send the message
 required SendSpeedRqstResponse(<r>)

end

# Maniipulators used to execute body
interface # internal unnamed interface

 manipulator ChooseResponse()
  ordered
   # I don't have to put the preference consideration
   #  here, it could be external
   decide consider new SimpleDecideProdSet
   required ProposeChoices()
  end
 end

 manipulator ProposeChoices()
  required choice accept 1
   tag <self>.msg as new AcceptSpeedRequest()
```

```
    tag <self>.msg as new RejectSpeedRequest()
   end
  end

  manipulator SendSpeedRqstResponse(<r> isa
                                    SpeedRequestResponse)

   # This would be part of the general MessageManagement
   #  interface
   CreateReplyMessage(<self>.msg, <reply>)
   <reply>.contents.Insert(<r>.response)

   required decide consider new CommunicateMessage(<reply>)
  end

 end
end

###############
# Preference set referenced in above example
prodset SimpleDecideProdSet
 # Will reference this interface
 references interface IContactSpace

 # Local relation
 references contact isa Contact

 # Initializer
 init SimpleDecideProdSet(<c> isa Contact)
  consider <self>.contact <c>
 end

 # All preferences in this set
 preferences

  # If there is a contact ahead of me, prefer to
  #  reject over accept
  preference RejectWhenBlocked(<a> isa AcceptSpeedRequest,
                               <r> isa RejectSpeedRequest)
  precond
   considered <a>
   considered <r>
   MessageToContact(<a>, <r>)
   ContactAhead(<c>)  # Part of the IContactSpace interface
  then
   prefer commit <r> over <a>
  end
```

```
  #  default preference RejectWhenBlocked(
  default preference PreferAcceptSpeedRequest(
                                  <a> isa AcceptSpeedRequest,
                                  <r> isa RejectSpeedRequest)
  precond
   considered <a>
   considered <r>
  then
   prefer commit <a> over <r>
  end
 end

 interface  # An unnamed, internal interface

  # The complex logic for determining what accept
  #  and reject tags refer to
  query MessageToContact(<a> isa AcceptSpeedRequest,
                         <r> isa RejectSpeedRequest)
   exists <a> tagging <m>
   exists <r> tagging <m>
   <m>.from <self>.contact
  end

 end

end
```

Appendix D : Relevant Soar Code for the AMBR Example

The complete Soar code listing is provided as a separate deliverable. This excerpt below demonstrates the solution approach, but excludes much of the runtime library code necessary to make this code a fully functional Soar program.

```
sp [compiled*manipulator-choose-response*propose*accept-speed-
request
   (state <s> ^name manipulator-propose-choices
              ^transform <t>
              ^preference-sets.preference-set.contact <c>)
  -(<t> ^local-transform-memory.message.tagged-by.type speed-
request-response)
   -->
   (<s> ^operator <o> + )
   (<o> ^name accept-speed-request
        ^transform <t>
        ^contact <c>)
]
sp [compiled*apply*accept-speed-request*create-response
   (state <s> ^operator <o1>)
   (<o1> ^name accept-speed-request
         ^transform.local-transform-memory <loc>
         ^contact.id <id>)
   -->
   (<loc> ^response <res>)
   (<res> ^type speed-request-response
          ^value accept
          ^to <id>)
]

sp [compiled*apply*accept-speed-request*tag-message
   (state <s> ^operator <o1>)
   (<o1> ^name accept-speed-request
         ^transform.local-transform-memory <loc>)
   (<loc> ^message <msg>
          ^response <res>)
```

```
   -->
   (<msg> ^tagged-by <res>)
]
# Should really be more decisions here...
sp [compiled*manipulator-choose-
response*elaborate*substate*name*manipulator-choose-response
   (state <s>
              ^superstate.operator <so>)
   (<so> ^name manipulator-choose-response)
   -->
   (<s> ^name manipulator-propose-choices)
]


sp [compiled*manipulator-choose-
response*elaborate*substate*copy*transform
   (state <s> ^name manipulator-propose-choices
              ^superstate.operator <so>)
   (<so> ^transform <t>)
   -->
   (<s> ^transform <t>)
]

sp [compiled*manipulator-choose-response*propose*reject-speed-
request
   (state <s> ^name manipulator-propose-choices
              ^transform <t>
              ^preference-sets.preference-set.contact <c>)
  -(<t> ^local-transform-memory.message.tagged-by.type speed-
request-response)
   -->
   (<s> ^operator <o> + )
   (<o> ^name reject-speed-request
        ^transform <t>
        ^contact <c>)
]

sp [compiled*apply*reject-speed-request*create-response
   (state <s> ^operator <o1>)
   (<o1> ^name reject-speed-request
         ^transform.local-transform-memory <loc>
         ^contact.id <id>)
   -->
   (<loc> ^response <res>)
   (<res> ^type speed-request-response
          ^value reject
          ^to <id>)
```

```
]


sp [compiled*apply*reject-speed-request*tag-message
   (state <s> ^operator <o1>)
   (<o1> ^name reject-speed-request
         ^transform.local-transform-memory <loc>)
   (<loc> ^message <msg>
          ^response <res>)
   -->
   (<msg> ^tagged-by <res>)
]

# I think we should delete the goal here but, for now, we'll
just mark it
# as reconsidered.  Whatever element of the compiler deals with
reconsidered
# goals, shouldnt propose to re-activate reconsidered
achievement goals
# marked as achieved.

sp [hlsr-rtl*asr-hlsr*propose*achievement-goal-achieved
   (state <s> ^name asr-hlsr
              ^goals.goal <g>)
   (<g> ^status achieved
        ^tags.hlsr-state activated)
   -->
   (<s> ^operator <o> + )
   (<o> ^name achievement-goal-achieved
        ^goal <g>)
]

# Make achieved persistent, to ensure we know this goal was
achieved at one time
sp [hlsr-rtl*apply*achievement-goal-achieved*save-status
   (state <s> ^operator <o1>)
   (<o1> ^name achievement-goal-achieved
         ^goal <g>)
   (<g> ^status achieved)
   -->
   (<g> ^status achieved)
]

sp [hlsr-rtl*apply*achievement-goal-achieved*reconsider-goal
   (state <s> ^operator <o1>)
   (<o1> ^name achievement-goal-achieved
         ^goal.tags <g>)
```

```
   (<g> ^hlsr-state activated)
   -->
   (<g> ^hlsr-state activated - reconsidered +)
]

# This operator proposes any new-considered goal on the state.
# make-new-goal operators should not be indifferent

# Invariants must hold in order to activate a goal
sp [hlsr-rtl*asr-hlsr*propose*make-new-goal
   (state <s> ^name asr-hlsr
              ^goals.goal <g>)
   (<g> ^tags.hlsr-state new-considered
        ^tags.invariants-hold t)
   -->
   (<s> ^operator <o> + )
   (<o> ^name make-new-goal
        ^goal <g>)
#Compiler would also fill in rest of template stuff here
]

# These productions dont appear to do anything, but they
# ensure that the goal will stick around, even if the activator
# elaboration no longer matches; they make goals persistent

# Want to provide o-support for ^goal
sp [hlsr-rtl*apply*make-new-goal*make-goal-persistent
   (state <s> ^operator <o1>
              ^goals <gs>)
   (<gs> ^goal <g>)
   (<o1> ^name make-new-goal
         ^goal <g>)
-->
   (<gs> ^goal <g>)
]

# Want to provide o-support to ea attr under ^goal
# Do *not* want generally to provide persistence under tags
sp [hlsr-rtl*apply*make-new-goal*make-goal-attrs-persistent
   (state <s> ^operator <o1>)
   (<o1> ^name make-new-goal
         ^goal <g>)
   (<g> ^<attr> <val>)
-->
   (<g> ^<attr> <val>)
]
```

```
# This should terminate the operator
sp [hlsr-rtl*apply*make-new-goal*hlsr-state*activated
   (state <s> ^operator <o1>)
   (<o1> ^name make-new-goal
         ^goal <g>)
   (<g> ^tags <t>)
   (<t> ^hlsr-state [ new-considered <new> ])
-->
   (<t> ^hlsr-state activated + <new> -)
]

# Computed via elaboration
#sp [hlsr-rtl*apply*make-new-goal*status*in-progress
#   (state <s> ^operator <o1>)
#   (<o1> ^type make-new-goal
#         ^goal <g>)
#-->
#   (<g> ^status in-progress)
#]
#

# This operator proposes any new-considered transform on the
state.
# make-new-transform operators should not be indifferent

# Invariants must hold in order to activate a goal
sp [hlsr-rtl*asr-hlsr*propose*make-new-transform
   (state <s> ^name asr-hlsr
              ^transforms.transform <g>)
   (<g> ^tags.hlsr-state new-considered
        ^tags.invariants-hold t)
   -->
   (<s> ^operator <o> + )
   (<o> ^name make-new-transform
        ^transform <g>)
#Compiler would also fill in rest of template stuff here
]

# These productions dont appear to do anything, but they
# ensure that the goal will stick around, even if the activator
# elaboration no longer matches; they make goals persistent

# Want to provide o-support for ^transform
sp [hlsr-rtl*apply*make-new-xform*make-xform-persistent
   (state <s> ^operator <o1>
              ^transforms <gs>)
   (<gs> ^transform <g>)
```

```
   (<o1> ^name make-new-transform
         ^transform <g>)
-->
   (<gs> ^transform <g>)
]

# Want to provide o-support to ea attr under ^transform
# Do *not* want generally to provide persistence under tags
sp [hlsr-rtl*apply*make-new-transform*make-transform-attrs-
persistent
   (state <s> ^operator <o1>)
   (<o1> ^name make-new-transform
         ^transform <g>)
   (<g> ^<attr> <val>)
-->
   (<g> ^<attr> <val>)
]

# This should terminate the operator
sp [hlsr-rtl*apply*make-new-transform*hlsr-state*activated
   (state <s> ^operator <o1>)
   (<o1> ^name make-new-transform
         ^transform <g>)
   (<g> ^tags <t>)
   (<t> ^hlsr-state [ new-considered <new> ])
-->
   (<t> ^hlsr-state activated + <new> -)
]

# This manipulator does two things:
# 1: activates the SimpleDecidePrefSet
# 2: executes the ProposeChoices() manipulator

# These steps are ordered so I am assuming the
# activation of the pref set will set a tag on the
# transform.  In general, the decision to allow
# one manipulator to execute another looks wrong
# to me, but Ill go forward for now, using an ONC.

# The name of the statement and the name of the
# operator should be the same!
sp [compiled*asr-hlsr*propose*manip-choose-response
   (state <s> ^name asr-hlsr
              ^transforms.transform <t>)
   (<t> ^name simple-answer-decision
        ^status executing
        ^tags <tg>)
```

```
    (<tg> ^hlsr-state activated
          ^required-statement <r>)
    (<r> ^name manipulator-choose-response
        -^status completed)
    -->
    (<s> ^operator <o> + )
    (<o> ^name manipulator-choose-response
         ^transform <t>)
]

sp [compiled*apply*manip-choose-response*status*executing
    (state <s> ^operator <o1>)
    (<o1> ^name manipulator-choose-response
          ^transform <t>)
    (<t> ^tags.required-statement <r>)
    (<r> ^name manipulator-choose-response
         ^status not-yet-executed)
    -->
    (<r> ^status not-yet-executed - executing +)
]



# Create the preference set to help make the decisions...
# ... put contact in here for now.
sp [compiled*apply*manip-choose-response*activate*pref-
set*SimpleDecide
    (state <s> ^operator <o1>
               ^preference-sets <p>
               ^top-state.global-objects.contact <c>)
    (<o1> ^name manipulator-choose-response
          ^transform <t>)
    (<c> ^id <name>)
    (<t> ^goal.message.from <name>)
   - [ (<p> ^preference-set <p1>)
       (<p1> ^name simple-decide-prefset) ]
    -->
    (<p> ^preference-set <p-new>)
    (<p-new> ^name simple-decide-prefset
             ^transform <t>
             ^preferences <pf>
             ^contact <c>
             ^tags <tg>)
    (<tg> ^hlsr-state activated)
    (<pf> ^preference <p1> <p2>)
    (<p1> ^name accept-when-clear)
    (<p2> ^name reject-when-blocked          )
```

```
]


# I think this might be a hack -- Im basically terminating the
# manipulator when the response is generated.  But that occurs
# in propose choices.  Generally, we may need to represent
# required statements in manipulators declaratively, as for
xforms
sp [compiled*apply*manip-choose-response*status*complete
   (state <s> ^operator <o1>)
   (<o1> ^name manipulator-choose-response
         ^transform <t>)
   (<t> ^tags.required-statement <r>
        ^local-transform-memory.message.tagged-by.type speed-
request-response)
   (<r> ^name manipulator-choose-response
        ^status executing)
   -->
   (<r> ^status executing - completed +)
]
# Im not sure if this is smart or a hack.  It seems like there
# are going to be lots of cases where we just want to check
# that some HLSR-level has been taken.  This operator checks
# an attribute in the transform local memory to determine if
# it has been tagged with an object of the type specified
# in required statement.  It might be easier to make this
# come from the compiler, but I think that could lead to many,
# many rules where these should suffice -- well see

sp [hlsr-rtl*asr-hlsr*propose*manipulator-confirm-local-object-
tagged-with
   (state <s> ^name asr-hlsr
              ^transforms.transform <t>)
   (<t> ^tags <tg>
        ^local-transform-memory.<attr>.tagged-by.type <tag-
type>)
   (<tg> ^hlsr-state activated
         ^required-statement <r>
         )
   (<r> ^status not-yet-executed ^type manipulator-confirm-
local-object-tagged-with
         ^attribute-name <attr>
         ^tagged-by <tag-type>)
   -->
   (<s> ^operator <o> +)
   (<o> ^name manipulator-confirm-local-object-tagged-with
        ^required-statement <r>
```

```
        ^transform <t>
                   )
]

# Only action of this operator is to mark the required statement
completed
sp [hlsr-rtl*apply*manipulator-confirm-local-object-tagged-with
   (state <s> ^operator <o1>
              )
   (<o1> ^name manipulator-confirm-local-object-tagged-with
         ^required-statement <r>)
   (<r> ^status not-yet-executed)
   -->
   (<r> ^status not-yet-executed - completed +)
]

# If one of these is competing with a regular manipulator, just
execute this one
# first and get it out of the way

# Uses assumption that name of non-confirm ops are the same as
required statements -- for confirm
# ops, only the required statement type will match op name
sp [hlsr-rtl*asr-hlsr*compare*manipulator-confirm-local-object-
tagged-with
   (state <s> ^name asr-hlsr
              ^operator <o1> +
                        <o2> +
              ^transforms.transform <t>)
   (<t> ^tags <tg>)
   (<tg> ^hlsr-state activated
         ^required-statement <r2>)
   (<r2> ^name <name>
         ^status not-yet-executed)
   (<o1> ^name manipulator-confirm-local-object-tagged-with)
   (<o2> ^name <name>)
   -->
   (<s> ^operator <o1> > <o2>)
]

# Some of this manipulator will be skipped... dont want to go
thru details of sending message


# this manipulator requires a response on the local transform
state
```

```
sp [compiled*asr-hlsr*propose*manipulator-send-speed-request-
response
   (state <s> ^name asr-hlsr
              ^transforms.transform <t>)
   (<t> ^name simple-answer-decision
        ^local-transform-memory <l>
        ^tags <tg>)
   (<l> ^response <res>)
   (<res> ^type speed-request-response)
   (<tg> ^hlsr-state activated
         ^required-statement <r>)
   (<r> ^name manipulator-send-speed-request-response
        -^status completed)
   -->
   (<s> ^operator <o> +)
   (<o> ^name manipulator-send-speed-request-response
        ^transform <t>)
]


# Not really needed for this implementation but would be if not
for simplifications.
sp [compiled*apply*manipulator-send-speed-request-
response*status*executing
   (state <s> ^operator <o1>)
   (<o1> ^name manipulator-send-speed-request-response
         ^transform <t>)
   (<t> ^tags.required-statement <r>)
   (<r> ^name manipulator-send-speed-request-response
        ^status not-yet-executed)
   -->
   (<r> ^status not-yet-executed - executing +)
]


sp [compiled*apply*manipulator-send-speed-request-response*new-
goal*send-response
   (state <s> ^operator <o1>
              ^goals <g>)
   (<o1> ^name manipulator-send-speed-request-response
         ^transform <t>)
   (<t> ^local-transform-memory.response <r>
        ^goal <supergoal>)
   -->
   (<g> ^goal <g-new>)
   (<g-new> ^name send-message
            ^goal-type achievement-goal
```

```
            ^outgoing-message <r>
            ^supergoal <supergoal>
            ^tags <tg>)
   (<tg> ^hlsr-state new-considered
         ^created-by <t>)
]


sp [compiled*apply*manipulator-send-speed-request-
response*status*completed
   (state <s> ^operator <o1>)
   (<o1> ^name manipulator-send-speed-request-response
         ^transform <t>)
   (<t> ^tags.required-statement <r>)
   (<r> ^name manipulator-send-speed-request-response
        ^status executing)
   -->
   (<r> ^status executing - completed +)
]
sp [compiled*asr-hlsr*propose*manipulator-tag-message-sent
   (state <s> ^name asr-hlsr
              ^transforms.transform <t>)
   (<t> ^name send-message
        ^status executing
        ^tags <tg>)
   (<tg> ^hlsr-state activated
         ^required-statement <r>)
   (<r> ^name manipulator-tag-message-sent
       -^status completed)
   -->
   (<s> ^operator <o> +)
   (<o> ^name manipulator-tag-message-sent
        ^transform <t>)
]


sp [compiled*apply*manip-tag-message-sent-
response*status*executing
   (state <s> ^operator <o1>)
   (<o1> ^name manipulator-tag-message-sent
         ^transform <t>)
   (<t> ^tags.required-statement <r>)
   (<r> ^name manipulator-tag-message-sent
        ^status not-yet-executed)
   -->
   (<r> ^status not-yet-executed - executing +)
]
```

```
# Create the preference set to help make the decisions...
sp [compiled*apply*manip-tag-message-sent*tag-message
   (state <s> ^operator <o1>)
   (<o1> ^name manipulator-tag-message-sent
         ^transform <t>)
   (<t> ^local-transform-memory.outgoing-message <m>)
   -->
   (<m> ^tagged message-sent)
]

sp [simulated*apply*manip-tag-message-sent*output-link
   (state <s> ^operator <o1>)
   (<o1> ^name manipulator-tag-message-sent
         ^transform <t>)
   (<t> ^local-transform-memory.outgoing-message <m>)
   -->
   (<s> ^sim-output-link.message <m>)
]

# I think this might be a hack -- Im basically terminating the
# manipulator when the response is generated.  But that occurs
# in propose choices.  Generally, we may need to represent
# required statements in manipulators declaratively, as for
xforms
sp [compiled*apply*manip-tag-message-sent*status*complete
   (state <s> ^operator <o1>)
   (<o1> ^name manipulator-tag-message-sent
         ^transform <t>)
   (<t> ^tags.required-statement <r>
        ^local-transform-memory.outgoing-message.tagged message-
sent
                                                          )
   (<r> ^name manipulator-tag-message-sent
        ^status executing)
   -->
   (<r> ^status executing - completed +)
]

# part of activator.  When an invariant in an activator object
no longer holds,
# the activated object should be reconsidered.

# I could make these 1 production but it's simpler just to put
them into different
# productions for now.  I dont we'll need to access the object
other than the
```

```
# hlsr-state, so I just include a pointer to the object here.

sp [hlsr-rtl*asr-hlsr*propose*reconsider-goal-when-invariants-
no-longer-hold
   (state <s> ^name asr-hlsr
              ^goals.goal <obj>)
   (<obj> ^tags <t>)
   (<t> ^hlsr-state activated
        -^invariants-hold t)
   -->
   (<s> ^operator <o> + =)
   (<o> ^name reconsider-object-when-invariants-no-longer-hold
        ^goal <obj>)
]

sp [hlsr-rtl*asr-hlsr*propose*reconsider-transform-when-
invariants-no-longer-hold
   (state <s> ^name asr-hlsr
              ^transforms.transform <obj>)
   (<obj> ^tags <t>)
   (<t> ^hlsr-state activated
        -^invariants-hold t)
   -->
   (<s> ^operator <o> + =)
   (<o> ^name reconsider-object-when-invariants-no-longer-hold
        ^transform <obj>)
]

# Should be the same for any of goals, transforms, beliefs,
preferences
sp [hlsr-rtl*apply*reconsider-object-when-invariants-no-longer-
hold*hlsr-state*reconsidered
   (state <s> ^operator <o1>)
   (<o1> ^name reconsider-object-when-invariants-no-longer-hold
         ^<object>.tags <t>)
   (<t> ^hlsr-state <active>)
   -->
   (<t> ^hlsr-state <active> - reconsidered +)
]

sp [hlsr-rtl*asr-hlsr*propose*reconsider-transform-when-
completed
   (state <s> ^name asr-hlsr
              ^transforms.transform <t>)
   (<t> ^tags.hlsr-state activated
        ^status completed)
   -->
```

```
   (<s> ^operator <o> + )
   (<o> ^name reconsider-transform-when-completed
        ^transform <t>)
]

sp [hlsr-rtl*apply*reconsider-transform-when-completed*make-
status-persistent
   (state <s> ^operator <o1>
              )
   (<o1> ^name reconsider-transform-when-completed
         ^transform <t>)
   (<t> ^status completed)
   -->
   (<t> ^status completed)
]


sp [hlsr-rtl*apply*reconsider-transform-when-
completed*reconsider-transform
   (state <s> ^operator <o1>
              )
   (<o1> ^name reconsider-transform-when-completed
         ^transform <t>)
   (<t> ^tags <tg>)
   (<tg> ^hlsr-state activated)
   -->
   (<tg> ^hlsr-state activated - reconsidered +)
]
```

Appendix E : Relevant ACT-R Code for the AMBR Example

```
#|
Goal(-related) types
|#

(chunk-type goal  ;;; single inheritance
       ;;; all chunks are named - no slot
       type   ;;; kind of goal - not subtype
       state  ;;; current status of goal
       parent ;;; previous or constituent
       ;;; local slots defined for each subtype
       tags ;;; only most recent)

(chunk-type tag  ;;; also subtypes of tags
       goal   ;;; tag points back to goal
       hlsr-state  ;;; current state)

(chunk-type (answer-speed-request (:include goal)))

;;; Achievement goals
;;; Any particular slots?  Assuming one slot called achieved

(chunk-type (achievement-goal (:include goal))
     achieved)

;;; Goal: AcceptRejectResponse
;;; Note: chunk/goal slots are not typed
;;; Interestingly, they were in ACT-RN for pattern decoding
purposes

 (chunk-type (accept-reject-response (:include achievement-
goal))
     msg)


#|
Chunk types:
Use the chunk type hierarchy  to reflect the object type
inheritance.
At this point no object has so many fields as to merit
decomposition into multiple chunk (types), though contact is
starting to get close.
```

```
|#

(chunk-type message from)
(chunk-type (request-speed-change-message (:include message))
(chunk-type entity)
(chunk-type (contact (:include entity))
 speed location color id screen-location)

#|
Transform-related chunks and chunk types
Again, the chunk type hierarchy is used, but also the
specification of default initial values for the init conditions.
|#

(chunk-type speed-request-response-options)

(add-dm
     (accept isa speed-request-response-options)
     (reject isa speed-request-response-options))

(chunk-type speed-request-response response)

(chunk-type (accept-speed-request (:include request-response)))
     (response "accept"))

(chunk-type (reject-speed-request (:include request-response)))
     (response "reject"))


#|
Activators:
Decomposes into two productions:
One to generate and focus on the goal
The second to generate the tag
|#

 (p propose-goal-answer-speed-request-create
     =goal>
            isa goal
     =retrieval>
            isa request-speed-change-message
            from =from
==>
     +retrieval>
            isa tag
            hlsr-state new-considered
            invariants-hold t
```

```
      +goal>     ;;;  Creates a new goal
             isa answer-speed-request
             type achievement-goal
             message =retrieval
             tags initialize)

 (p propose-goal-answer-speed-request-tag
      =goal>
             isa answer-speed-request
             tags initialize
      =retrieval>
             isa tag
             hlsr-state new-considered
             invariants-hold t
==>
       =retrieval>  ;;;Creates new tag pointing to goal
             goal =goal
       =goal>               ;;; Goal also points to tag
             tags =retrieval)

#|
Interfaces can be represented as productions detecting specific
sets of conditions
As usual, because of the conditions on memory access, those
tests will involve buffer contents.  It is left to the
discretion of the compiler how those contents got there.
For instance, if an explicit retrieval is attempted in a typical
retrieve-test production pair, then that retrieval will take
time and trigger questions of plausibility since it is not just
a passive test but an active one.  If on the other hand the
interface consists of only the test, then it depends on some
other productions to field the buffer with the correct content,
and thus is not an independent piece of behavior.  We will go
this way here however.
|#

 (p internal-is-met
     =goal>
      isa accept-reject-response
             msg =msg
     =retrieval>
      isa message
      reply =msg
      status sent
==>
    =goal>
      achieved t)
```

```
#|
We will represent failure here as failure to perform a past
retrieval, such as retrieving the reply message tested in
InternalIsMet condition.
|#

(p failed
    =goal>
     isa accept-reject-response
     msg =msg
   =retrieval>
     isa failure
==>
    =goal>
     achieved failed)

#|
Transforms correspond to sets of productions.  They could be
grouped around a particular goal created to that effect.  For
illustration and differentiation, we will define them here as
applying to a general goal with a specific slot value indicating
the transfer in progress.
An activator then becomes a production that sets that slot value
to trigger the tranform productions matching that slot-value
pattern.  Here we assume that a special slot called transform is
used to that effect.
|#

(p propose-transform-simple-answer-decision
    =goal>
     isa accept-reject-response
     transform nil
==>
    =goal>
     transform simple-answer-decision)

(p simple-answer-decision
    =goal>
     isa accept-reject-response
     msg =msg
     transform simple-answer-decision
==>
    +retrieval>
     isa message
     reply =msg
    +goal>
```

```
      isa choose-response
      reply =retrieval)

(p send-speed-request-response
    =goal>
      isa accept-reject-response
      msg =msg
   =retrieval>
      isa message
      reply =msg
==>
    +goal>
      isa send-speed-request-response
      response =retrieval)

#|
The same reasoning as applied to transforms could apply to
manipulators.
In this case, both for illustration and differentiation, we will
represent the manipulator as a specific goal with associated
productions.
|#

(p accept-speed-request
     =goal>
     isa choose-response
     reply =retrieval
     =retrieval>
     isa message
     reply =msg
     response nil
==>
     =retrieval>
     response "accept")

(p reject-speed-request
     =goal>
     isa choose-response
     reply =retrieval
     =retrieval>
     isa message
     reply =msg
     response nil
==>
     =retrieval>
     response "reject")
```

```
(p send-speed-request-response
     =goal>
     isa send-speed-request-response
     response =retrieval
     =retrieval>
     isa message
     reply =msg
     response =response
==>
      +goal>
     isa reply
     content =response)



#|
Preferences can be represented as setting production utilities,
which arbitrate in conflict resolution, but this requires that
they be independent of other considerations, such as conditions
on values.  Otherwise, they need to be explicitly represented as
productions steering conflict resolution one way or another.
Here, we will illustrate both: utilities to prefer accept over
reject, but a special production to test for reject condition.
|#

(spp accept-speed-request :p 0.6)

(spp reject-speed-request :p 0.4)

(p condition-reject-speed-request
     =goal>
     isa choose-response
     reply =retrieval
     =retrieval>
     isa message
     reply =msg
     response nil
     =visual>
     isa contact
     position ahead
==>
     =retrieval>
     response "reject")
```